Embedding-based Automated Assessment of Domain Models

Kua Chen

Department of Electrical and Computer Engineering

McGill University, Montreal

August, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

Domain modeling is an essential part of software engineering and serves as a way to represent and understand the concepts and relationships in a problem domain. Typically, students learn domain modeling by interpreting natural language problem descriptions and manually translating them into a domain model such as a Unified Modeling Language (UML) class diagram. Instructors evaluate these student-generated diagrams manually, comparing them against a reference solution and providing feedback. However, as enrollment in software engineering courses continues to rise, the manual grading of numerous student submissions becomes an overwhelming and time-intensive task for instructors. Recently, Large Language Models (LLMs) have exhibited impressive ability in domain understanding capabilities. Due to the lack of automated assessment, many automated modeling approaches rely on the manual evaluation of domain models during research on the modeling ability of LLMs. The manual evaluation process is time-consuming and hinders further related research to match the progress of ever-changing LLMs. Therefore, there is a need for automated assessment of domain models which alleviates instructors of the burdensome grading process and benefits automated modeling research. In this thesis, we propose a novel embedding-based approach that automatizes the assessment of domain models in a textual domain-specific language, against reference solutions created by modeling experts. The proposed algorithm showcases remarkable proficiency in matching model elements across domain models, achieving an impressive F_1-score of 0.82 for class matching, 0.75 for attribute matching, and 0.80 for relationship matching. Furthermore, our algorithm yields grades highly correlated with manual assessments, with correlations exceeding 0.8 and mean absolute errors below 0.05.

However, of the statistical tests related to grades, only those related to attributes show no statistically significant difference when compared to manual grading. In contrast, those related to classes and relationships reveal statistically significant differences, highlighting areas for potential improvement.

Résumé

La modélisation de domaine est une partie essentielle du génie logiciel et sert de moyen de représenter et comprendre les concepts et relations dans un domaine problématique. Typiquement, les étudiants apprennent la modélisation de domaine en interprétant des descriptions de problèmes en langage naturel et en les traduisant manuellement en un modèle de domaine tel qu'un diagramme de classes UML. Les instructeurs évaluent ces diagrammes générés par les étudiants manuellement, en les comparant à une solution de référence et en fournissant des commentaires. Cependant, avec l'augmentation des inscriptions aux cours de génie logiciel, la notation manuelle de nombreuses soumissions d'étudiants devient une tâche écrasante et intensive en temps pour les instructeurs. Récemment, les grands modèles de langage (LLM) ont montré une capacité impressionnante à comprendre les domaines. En raison du manque d'évaluation automatisée, de nombreuses approches de modélisation automatisée reposent sur l'évaluation manuelle des modèles de domaine lors de la recherche sur la capacité de modélisation des LLM. Le processus d'évaluation manuelle est chronophage et entrave la recherche connexe pour suivre les progrès des LLM en constante évolution. Par conséquent, il est nécessaire d'avoir une évaluation automatisée des modèles de domaine qui soulage les instructeurs de la tâche fastidieuse de notation et profite à la recherche sur la modélisation automatisée. Dans cette thèse, nous proposons une approche novatrice basée sur l'incorporation qui automatise l'évaluation des modèles de domaine dans un langage spécifique au domaine textuel, par rapport à des solutions de référence créées par des experts en modélisation. L'algorithme proposé démontre une compétence remarquable dans la mise en correspondance des éléments de modèle entre les modèles de domaine, atteignant un

impressionnant score F_1 de 0.82 pour la mise en correspondance de classes, 0.75 pour la mise en correspondance d'attributs et 0.80 pour la mise en correspondance de relations. De plus, notre algorithme produit des notes fortement corrélées avec les évaluations manuelles, avec des corrélations dépassant 0.8 et des erreurs absolues moyennes inférieures à 0.05. Cependant, parmi les tests statistiques liés aux notes, seuls ceux concernant les attributs ne montrent aucune différence significative par rapport à la notation manuelle. En revanche, ceux concernant les classes et les relations révèlent des différences statistiquement significatives, soulignant des possibilités d'amélioration.

Acknowledgements

I want to extend my deepest gratitude to my respected supervisors, Professor Gunter Mussbacher and Professor Dániel Varró, whose generous support, valuable guidance, and insightful feedback have been pivotal throughout my master's studies. I have learned about them since my undergraduate years, and it is truly an honor to have them as my supervisors during my graduate studies. Their expertise and encouragement have profoundly shaped my research and refined my ideas.

I also want to thank my dedicated lab mates and co-workers: Boqi Chen and Yujing Yang. Boqi Chen is a Ph.D. candidate who has been a wonderful mentor in shaping my academic life. Yujing Yang is a master's student and a friend with whom I share a longstanding friendship. Their assistance and collaboration were essential during my research. Their ongoing support was vital in sustaining my motivation and focus throughout this journey.

My heartfelt appreciation extends to our esteemed industrial collaborator, Dr. Amir Feizpour, the CEO and co-founder of Aggregate Intellect, and to our funding source, Mitacs Canada, for their generous financial support. Without their generous funding, this work would not have been possible.

Lastly, I am sincerely grateful to my beloved family and my girlfriend, Qinyi Wang. Their unconditional love, understanding, and support have been the bedrock of my academic pursuit. Their faithful encouragement and belief in me have fueled my determination and drove me forward.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

EBNF	Extended Backus-Naur Form
FN	False Negative
FP	False Positive
GED	Graph Edit Distance
IR	Information Retrieval
KG	Knowledge Graph
LLMs	Large Language Models
MAE	Mean Absolute Error
MDE	Model-driven Engineering
ML	Machine Learning
NLP	Natural Language Processing
TN	True Negative
TP	True Positive
UML	Unified Modeling Language
NER	Named Entity Recognition

Chapter 1

Introduction

This chapter introduces the thesis starting by stating the context and motivation of domain modeling and domain model assessment in Section 1.1 and then describing the proposed research questions, objectives, and contributions for this thesis in Section 1.2. In the end, this chapter will provide an overview of the organization of the thesis in Section 1.3.

1.1 Context and Motivation

Domain modeling is a core process in software engineering that builds a domain model from various sources of information, including documents, stakeholder interactions, etc., to encapsulate the intricacies of a system into a coherent representation. A domain model offers a high-level abstraction of the system's structure to support the system's behavior and functionality tailored to the specific requirements of the target domain. It is represented as a UML class diagram [3] which typically comprises a set of system classes, a set of attributes of every class, and has relationships between classes but excludes any operations in classes. These relationships provide valuable insights into the interactions within the system. Classes, attributes, and relationships are called model elements. A typical software development life cycle begins with the design of a domain model with various sources of information. This domain model acts as a blueprint for the software. Developers then utilize code generation

tools to create skeleton code in Java or other programming languages based on the domain model, providing a foundation for implementing the business logic.

Software engineering students usually learn domain modeling by interpreting natural language problem descriptions and manually building a domain model by incrementally combining model elements together. These domain models need to be graded so that students can enhance their modeling ability. In university undergraduate-level modeling exercises, instructors play an essential role in assessing domain models created by students. Typically, the course instructor creates a reference domain model, against which each student submission is graded. The grading process is usually a matching process. Course instructors usually attempt to identify the occurrence of model elements in the reference model presented within the student's model. As the number of students taking software modeling courses increases, the grading gradually becomes an overwhelming workload for course instructors. This serves as the first motivation for researching on automated domain model assessment.

The second motivation for developing the automated domain model assessment is to facilitate research progress on domain modeling. In the research of domain modeling, most researchers have to evaluate the domain models generated from their proposed techniques. The manual evaluation is typically time-consuming and highly dependent on human expertise. This evaluation process limits the number of samples in the evaluation and requires researchers to dedicate a significant amount of time for manually evaluating the quality of generated models. This problem is even more significant when Large Language Models (LLMs) are used for automated domain modeling. LLMs have recently become a hot topic of many research areas since they demonstrated remarkable proficiency in language comprehension, generation, reasoning tasks, and especially powerful generalizability to tasks beyond natural language processing [4]. LLMs can perform different tasks without supervised training on the specific task using carefully designed input (called *prompt*). Using different prompt engineering [5] techniques, LLMs can achieve impressive performance on different tasks by only using a few labeled examples in the prompt. Their advanced capabilities represent a significant advancement in natural language processing, offering new

opportunities for applications in domain modeling. Extensive research has been denoted to investigate the modeling ability of LLMs and researchers can generate domain models more easily and rapidly with the help from LLMs [2, 6, 7]. However, they usually have to spend a large amount of time in evaluating the generated domain model manually due to a lack of a proper automated assessment of generated domain models. This issue hinders the research progress on investigating the modeling ability of LLMs. For example, in our previous research attempt [2], we employed a matching-based evaluation scheme to manually evaluate domain models generated by LLMs. The tedious evaluation process prevented us from investigating other directions of using LLMs due to time constraints. The aforementioned observations emphasize the necessity for an automated domain model assessment approach.

Before the era of LLMs, many approaches have attempted to automate the process of evaluating domain models to mitigate this intensive manual effort [8–15]. However, these approaches typically have one or two main limitations: (1) they require significant technical debt to implement, or (2) these approaches typically require the input model in a specific format, such as Ecore files. A domain model can be expressed by graphical or textual modeling languages such as UML [3], Umple [16], and Ecore [17]. These languages have strict syntax rules for representing various model elements. However, students may make syntactic mistakes while still being correct in the rest of the model, while existing evaluation approaches do not work for these scenarios. Similarly, because LLMs are trained on a large corpus of text, it is difficult to constrain them to adhere to such strict syntax, particularly when modeling examples may infrequently appear in the training set. As such, no existing approaches can smoothly perform fully automated domain model assessments to incorporate the era of LLM-empowered research on domain modeling.

Meanwhile, existing automated domain model assessment approaches rely on rule-based techniques for grading, while the machine-learning-based approaches applied are rather rare. One reason behind this is the lack of properly labeled training examples. Additionally, any domain model can be represented as graphs (not only the ones already in graphical notation).

Graph comparison techniques are widely used to compare simple and complex graphs. As a result, graph comparison techniques have been utilized for assessing domain models.

Problem Statement. This thesis aims to address the problem of fully automated domain model assessment. It aims to derive a complete domain model assessment algorithm from plain text input without any human interaction or supervised training while aiming to reduce technical debt.

1.2 Research Questions, Objectives, and Contributions

To assist course instructors in reducing the grading workload for domain models in educational exercises, and to advance the research progress of LLM-empowered domain modeling, this thesis proposes a novel algorithm for automatically assessing a domain model against a reference domain model. The proposed algorithm takes two models as inputs: the candidate domain model to be assessed and the reference model which is usually created by modeling experts. The algorithm iterates over all the elements in both models and performs graph-based matching and checks. The proposed algorithm can automatically match model elements between two models, assign a score to each element, and generate a final grade for the candidate model to provide insightful assessments of domain models.

The matching process relies on text embedding and graph comparison. Different word embedding and sentence embedding techniques are employed to identify similarities between model elements. The algorithm determines cosine similarities between each model element pair using embeddings and subsequently identifies the optimal graph edit distance to match elements. Once a match is established, the algorithm assigns a grade based on pre-defined rules.

The algorithm employs a multi-stage process to match classes, attributes, and relationships separately. Producing suitable matches for elements is essential for generating meaningful assessments of domain models. After the matching process, the algorithm calculates

metrics for each type of model element and generates a final grade using weighted averages of the metrics. This leads to the proposed research questions.

Research Questions

The fundamental mechanism in the proposed algorithm is to match model elements between two domain models. Without a reasonable element matching, the algorithm would not be able to produce grades that truly reflect the correctness of the assessed domain model. Therefore, it is necessary to investigate whether the algorithm can match elements correctly.

RQ 1

What is the performance of the algorithm in matching a candidate domain model to a reference model regarding classes, attributes, and relationships?

After the element matching process, the algorithm generates precision, recall, F_1-scores, and a final grade for each assessed domain model. Therefore, this thesis also wants to investigate whether these generated statistics align with human-generated statistics.

RQ 2

To what extent do the algorithm-generated statistics compare with those produced by human grading?

Objectives

The objective of this thesis is to propose and evaluate the feasibility of fully automating domain modeling assessment using the proposed algorithm. Specifically, this thesis evaluates the algorithm's performance in matching model elements and generating grades for domain models. Additionally, this thesis aims to identify both the advantages and limitations of the approach for fully automated domain model assessment.

Contributions

Given two domain models, this thesis presents a novel approach for fully automated domain model assessment using embeddings and graph comparison techniques. Compared to existing methodologies, the proposed algorithm leverages word embeddings, sentence embeddings and graph matching algorithms to match model elements, enabling automated domain model assessment. It assigns scores to each model element and calculates precision, recall, and F_1-scores separately for classes, attributes, and relations. To the best of my knowledge, this is the first kind of approach which applies a combination of embeddings and graph matching for automated assessment of domain models. The specific contributions of this thesis are the following:

- This thesis proposes a novel fully automated model element matching pipeline by framing the automated domain model assessment as a graph matching problem.
- This thesis proposes a model based on hash maps for storing the matching information generated by the algorithm.
- This thesis provides a new data set of 20 real student solutions for evaluating automated domain model assessment. The data set includes detailed matching information of each student solution, which can be used for validating the algorithm's performance.
- An automated evaluation pipeline of comparing modeling matches is proposed and utilized.
- This thesis conducts an experiment using 20 real student solutions to analyze the precision, recall, and F_1-scores of the algorithm.

1.3 Thesis Organization

This section provides an overview of the organization of the thesis.

- **Chapter 2:** This chapter provides background information and defines fundamental concepts used throughout the thesis. This chapter explains domain model representation, embeddings, graph similarity measures, and evaluation metrics. I contribute to the full chapter except for the domain representation which is adopted from our previous publication [2], a joint work with all authors of the paper. This paper conducts a comprehensive investigation into the modeling capabilities of contemporary LLMs and examines how prompts influence their effectiveness in generating domain models. The co-authors of this paper include the supervisors of this thesis, Professor Gunter Mussbacher and Professor Dániel Varró, as well as Yujing Yang, a master's student, and Boqi Chen, a PhD candidate, both of whom are students at McGill University and share the same supervisors with me. Another co-author, José Antonio Hernández López, is a PhD candidate who has a long-standing collaboration with our research group.

- **Chapter 3:** This chapter presents the proposed algorithm and explains each component in detail, including the matching mechanism with the usage of embedding and graph similarity measures. It also covers the scoring for each element and the calculation of final grades. The proposed algorithm is designed and implemented by the author of this thesis independently. I make a full contribution to this chapter.

- **Chapter 4** This chapter demonstrates the evaluation procedure of the algorithm, including the evaluation pipeline and calculation of metrics. The manual evaluation procedure is adopted from our joint work [2]. The evaluation pipeline of comparing matches and calculation of metrics are developed and written solely by the author of this thesis.

- **Chapter 5:** This chapter focuses on addressing proposed research questions and discusses the experimental setup, dataset preparation, and results concerning the algorithm. Towards the end, it also discusses threats to validity. The author makes a full contribution to this chapter.

- **Chapter 6:** This chapter presents research related to this thesis, including a survey of research on domain model assessment and Natural Language Processing (NLP) for Model-driven Engineering (MDE). Additionally, other research activities conducted during my master's studies are discussed, including a joint publication on NLP [18]. Some of the related work on domain model assessment and NLP for MDE are referenced from our joint publications [2, 7]. The author of this thesis contributes to the rest of this chapter.

- **Chapter 7:** This chapter presents a summary of the key accomplishments of the thesis and outlines future work. The author makes a full contribution to this chapter.

1.4 Summary

In conclusion, this chapter provides a comprehensive overview of the thesis. It begins by explaining the context and motivation of automated domain model assessment in Section 1.1, followed by the proposed research questions, objectives, and contributions in Section 1.2. Finally, the organizational structure of this thesis is demonstrated in Section 1.3. In the subsequent chapter, this thesis will delve into the fundamental concepts and techniques upon which the algorithm is founded.

Chapter 2

Background

In this chapter, we describe the foundations for the relevant research areas of this thesis. As a starting point, we define fundamental concepts and a representation of domain models in Section 2.1. Subsequently, we examine the essential topics of embeddings (Section 2.2) and graph similarity measures (Section 2.3), both of which form integral components of the proposed algorithm. These discussions aim to provide a comprehensive understanding of the building blocks essential for the development and application of the approach. To culminate this exploration, we introduce evaluation metrics in Section 2.4, explaining different levels of evaluation.

2.1 Domain Modeling

In domain modeling, engineers typically convert a textual domain specification into a domain model represented as a class diagram [3]. Domain modeling is a challenging and time-consuming task that requires expertise and experience. A domain model uses a subset of class diagram concepts to capture essential elements of a domain, their attributes, and their relationships but does not cover elements related to a more detailed design (e.g., operations and interfaces).

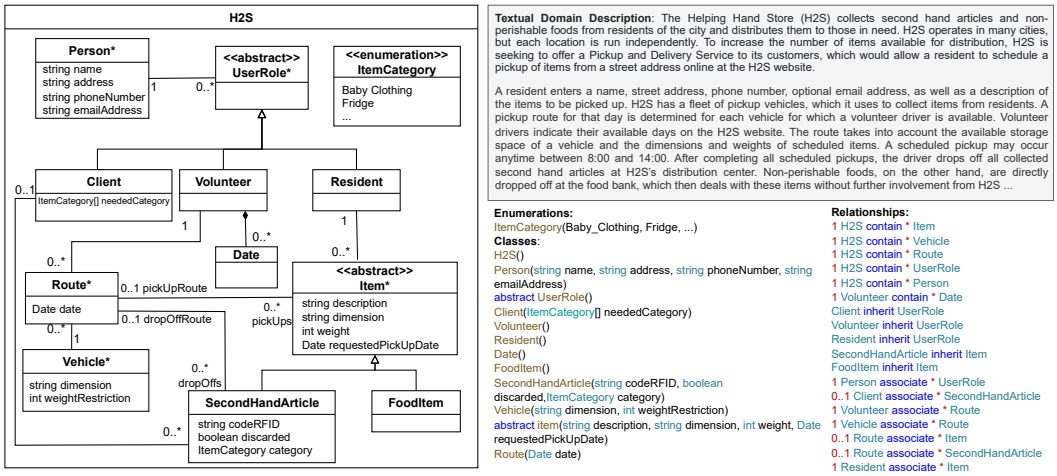


Figure 2.1: Example domain model (left) and its textual representation (bottom right) with the problem description (top right)

2.1.1 Domain Model Elements

Figure 2.1 shows an example domain model for a system called H2S on the left based on the problem description for which an excerpt is shown on the top right. It is a delivery and pickup system. This example covers the key concepts of domain models such as classes and enumerations (e.g., Person and ItemCategory, respectively), attributes (e.g., name), and three types of relationships among classes: a basic relationship called association (e.g., between Resident and Item), a whole-part relationship called composition (e.g., between Volunteer and Date), and an is-a relationship called generalization (e.g., between Item and FoodItem). An association relationship is represented as an undirected link in the diagram. A composition relationship is represented as a directed link with a diamond shape near the parent class. A generalization relationship is represented as a directed link with a triangle near the parent class. Classes may be abstract (i.e., they cannot be instantiated). Multiplicities are specified for associations/compositions and indicate how many instances of one class may be related to instances of the other class (e.g., 0..1, 0..*). Role names may be specified for the classes in an association/composition (e.g., pickUpRoute). Compositions may also be specified by placing classes inside a composite class (e.g., H2S). In that case, multiplicities are shown next to the name of the contained class (e.g., Person*). Association classes and

```
<class-diagram> ::= [<enumerations>] <classes> <relationships>
<enumerations> ::= "Enumerations: " (<enumeration>)+
<enumeration> ::= <string> "(" <literals> ")"
<literals> ::= <string> | <string> ", " <literals>
<classes> ::= "Classes: " (<class>)+
<class> ::= ["abstract"] <string>"(" [<attributes>] ")"
<attributes> ::= <attribute> | <attribute> ", " <attributes>
<attribute> ::= <type>["[]"] <string>
<relationships> ::= "Relationships: " [<composition>]* [<inheritance>]* [<association>]*
<composition> ::= <mul> <string> "contain" <mul> <string>
<inheritance> ::= <string> "inherit" <string>
<association> ::= <mul> <string> "associate" <mul> <string>
<type> ::= <string>
<mul> ::= "*" | <num> | <num>".."("*"|<num>)
```

Figure 2.2: EBNF format of a domain model

n-ary associations are excluded since they are not always supported (e.g., Ecore [17] does not support them).

2.1.2 Domain Model Representation

Our prior work on automated domain modeling [2] provides a comprehensive study on how LLMs generate domain models from textual problem descriptions and examines how prompts influence the quality of these models. The study finds that while LLMs demonstrate impressive modeling capabilities, they still fall short of matching the expertise of human professionals. Motivated by this work, instead of depicting a domain model using a graphical view or any other modeling language, the focus of this thesis lies in developing an algorithm built upon the textual domain model representation established in our previous publication [2]. There are many other graphical or textual modeling languages such as UML [3], Umple [16], and Ecore [17] to express a domain model. They typically include strict syntax rules and domain-specific languages. The proposed algorithm builds on top of text embeddings which are trained and are expected to be used in natural languages. The domain model representation is explained in detail in the following paragraphs.

Figure 2.2 presents our domain model representation in Extended Backus-Naur Form (EBNF) format [19]. The specification defines enumeration and two types of classes, regular and abstract, with the latter indicated by the keyword *abstract*. Enumeration literals are specified in brackets for each enumeration, while attributes (if any) can be multi-valued indicated by square brackets. Keywords identify three relationship types between two classes: *associate* for an association relationship, *inherit* for a generalization relationship, and *contain* for a composition relationship. Multiplicity is required and positioned before each class name for association and composition. We ignore the role names for associations and compositions in the representation for simplicity. Figure 2.1 (bottom right) shows the H2S example following the EBNF format.

2.2 Embeddings

Embedding is a means of representing objects like text, images, and audio as points in a continuous vector space where the locations of those points in space are semantically meaningful to Machine Learning (ML) algorithms [20]. In the context of this thesis, embedding is used to translate words and sentences, e.g., class names and relationships, into continuous vector space so that they can be used for finding and matching their counterparts by measuring cosine similarities between each pair.

2.2.1 Word Embeddings

Word embeddings, as the name suggests, is a means of representing individual words into a fixed-length vector space so that the computer can understand. It has been proven to be useful in many natural language processing tasks [21]. It is also useful in the context of encoding model elements into vectors. Many researchers have dedicated considerable efforts to the development of word embeddings which can be broadly classified into 2 distinct categories proposed by Almeida and Xexéo [21]. Prediction-based models leverage local data like a word's contextual information and bear a resemblance to neural language models.

Conversely, count-based models rely on global information, such as corpus-wide statistics including word counts and frequencies.

Count-based Methods

This type of embedding method is based on certain statistical measures of global word context co-occurrence counts and frequencies to derive word embeddings. These methods aim to capture the semantic relationships between words by analyzing their patterns of occurrence in a large corpus of text. Some examples of count-based methods include one-hot encoding, Term Frequency-Inverse Document Frequency (TF-IDF), Co-occurrence Matrix, Global Vectors (GloVe), etc. Among these methods, one of the most popular methods is GloVe proposed by Pennington et al. in 2014 [22].

The GloVe model is an unsupervised learning algorithm for obtaining vector embedding of words, leveraging global corpus statistics, specifically a word-word co-occurrence matrix. Firstly, let us define X as the word-word co-occurrence matrix and X_{ij} in this matrix is the number of times word j appears in the context of word i. The authors propose a cost function to learn word embedding with a global co-occurrence matrix:

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \left(w_i^T w_j + b_i + b_j - \log(X_{ij}) \right)^2 \tag{2.1}$$

where

- w_i and w_j are the word embeddings to be learned while minimizing the cost function.
- b_i and b_j are biases.
- V is the size of the vocabulary.
- $f(X_{ij})$ is a weighting function, giving importance to very frequent word pairs. It is defined as:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha} & \text{if } x \leq x_{\max} \\ 1 & \text{otherwise} \end{cases} \tag{2.2}$$

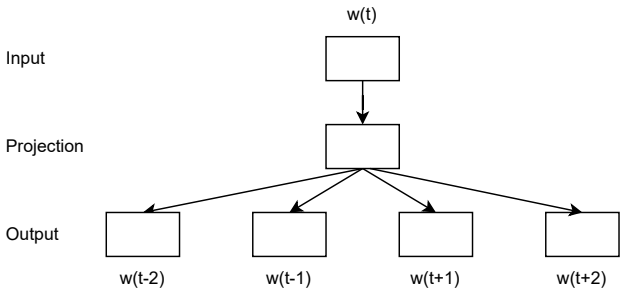


Figure 2.3: The skip-gram model architecture

where $x_{\max}$ is a predefined maximum co-occurrence count (often set empirically), and α is a weighting exponent (typically between 0.5 and 1.0) that controls the strength of the weighting.

Overall, the GloVe model learns word embeddings by minimizing the square error between the dot product of the word embeddings and the log of the co-occurrence counts X_{ij}, weighted by $f(X_{ij})$.

Prediction-based Methods

Prediction-based methods are deeply linked with neural networks and utilize their internal weights as the embeddings of the words. A word's embedding is just the projection of the raw word vector into the first layer of such models, the so-called embedding [21]. One of the well-known methods belonging to this category is Word2Vec which can be further divided into two specific techniques: continuous bag-of-words [23] and skip-gram [24]. For the former, its main goal is to predict the center target word given its surrounding words by training and using a neural network. For instance, given a sentence, *The fox jumps over the dog*, the word *jumps* can be masked and the neural network is trained to predict this word with the input *The fox _ over the dog*. The latter method, skip-gram, possesses a similar but flipped mechanism. Given a word, a neural network is trained to predict the surrounding words, as shown in Figure 2.3. We describe the mechanism in more detail in the following paragraphs.

The training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document [24]. Let us define a

window size of c and center word w_t. This means there are c words on the left and right of w_t, i.e., $W = [w_{t-c}...w_{t-1}, w_t, w_{t+1}, w_{t+c}]$. The objective of the Skip-gram model is to maximize the average log probability:

$$\max \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j}|w_t) \tag{2.3}$$

where T is the total number of the vocabulary and $p(w_{t+j}|w_t)$ means the probability of observing word w_{t+j} given the center word w_t. This conditional probability is often modeled using the softmax function:

$$p(w_O|w_I) = \frac{e^{u'^T_{w_O} \cdot u_{w_I}}}{\sum_{v=1}^{V} e^{u'^T_{w_v} \cdot v_{w_I}}} \tag{2.4}$$

where u and $u^{'}$ are vector representations of words.

This paragraph explains the implementation details of getting the word embedding in the Skip-gram model. There are two matrices U with dimension N by V and $U^{'}$ with dimension V by N, where N is the dimension of the embedding. Initially, the word w_t is converted to one-hot encoding into a vector x of dimension 1 by V. Then x is multiplied with the first matrix $U^{'}$ resulting in a vector y of dimension 1 by N. Then the vector y is multiplied with the second matrix U and produces the final output, a vector z of 1 by V. Each element in z represents a score or likelihood associated with each word in the vocabulary. Higher values typically indicate a higher likelihood that a particular word is appearing in the surrounding context of the input word w_t. The rows of matrix U and the columns of $U^{'}$ are the word embeddings. They are initialized randomly and updated during training with stochastic gradient descent. Eventually, a common approach for getting the final embedding is to take the average of both embeddings from the 2 matrices.

Both continuous-bag-of-words and skip-gram methods were originally pre-trained on general text sources like Wikipedia and Twitter. Therefore, when applying them to a specific domain, they may misinterpret some words in such domain. For example, the word state is usually associated with a country or nation, but in the domain of model-driven development (MDE), the state is more associated with state-machine. In 2023, Hernández et al. [25]

applied these two embedding methods with a large corpus of modeling texts and released WordE4MDE (Word Embeddings for MDE). They collected a corpus of MDE texts from well-known modeling conferences and journals and trained a skip-gram model as described above over this collected corpus. Since this skip-gram model specifically targets MDE, it is used in this thesis.

2.2.2 Sentence Embeddings

Similar to word embedding, sentence embedding refers to the process of representing a sentence as a fixed-size vector in a continuous vector space which a machine can understand. The goal is to capture the semantic meaning of the sentence in a way that allows comparisons and computations to be performed easily. One type of sentence embedding is averaging word embeddings. This type of method involves averaging or weighted averaging the word embeddings of all the words in a sentence. While these methods demonstrate high simplicity, they are unable to capture the relationships and nuances between words in a sentence. For instance, "My dog likes my cat" would be embedded the same as "My cat likes my dog". Therefore, these methods are unable to handle the change of order of words. Moreover, they are unable to consider the context of the word. The above-mentioned word embeddings are usually static. Thus the sentence embedding based on them would be static as well. However, the same word or the same sentence represents different meanings in different contexts, for example, in an ironic context.

Therefore, researchers introduced a more contextualized embedding approach. This well-known type of contextual sentence embedding method is pre-trained model embedding, typically empowered by transformer models like BERT (Bidirectional Encoder Representations from Transformers) [26] and GPT (Generative Pre-trained Transformer) [27]. These two major types of transformers show a lot of training processes in common. Both types of transformers are pre-trained on massive amounts of text data. The BERT models are pre-trained to learn to predict missing words in sentences (also known as masked language modeling), whereas GPT models are pre-trained to generate appending text (also known as

autoregressive language modeling). This pre-training process enables transformers to learn various presentations of words in different contexts. After pre-training processes, the transformer can be fine-tuned for specific downstream tasks such as text classification, information retrieval, etc to obtain better sentence embeddings. For the BERT model, when a sentence or sequence of tokens is fed into it, the [CLS] classification token is prepended to the input. These models are typically trained to have semantically similar sentence pairs to possess embeddings that are close together and dissimilar pairs to possess embeddings that are far apart. The output representation of the [CLS] token is used as the aggregated representation of the entire input sequence. This representation captures the semantic understanding of the entire sentence, considering the bidirectional context learned during pre-training. For GPT models, the last token in them is used as the sentence representation.

2.2.3 Cosine Similarity

Cosine similarity measures the similarity between two vectors of an inner product space [28]. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction [28]. The range of cosine similarity is from -1 to 1, where 1 reflects that two vectors are pointing in the same direction, and -1 represents that two vectors are pointing in exactly the opposite direction as shown in Figure 2.4. Cosine similarity is often used in data analysis for measuring the similarity of two items, with the help of embedding. Given a good embedding, texts that share similar meanings will have a larger cosine similarity value. For example, in information retrieval tasks, cosine similarity is often used as a key metric for retrieving documents whose embedding representations have a high similarity with that of the user input query.

Let us define n-dimensional vectors $\mathbf{A}$ and $\mathbf{B}$ mathematically as:

$$\mathbf{A} = [A_1, A_2, A_3, \ldots, A_n] \tag{2.5}$$

$$\mathbf{B} = [B_1, B_2, B_3, \ldots, B_n] \tag{2.6}$$

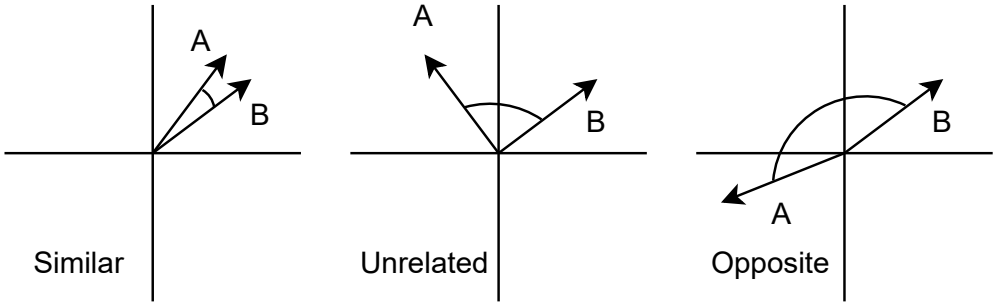


Figure 2.4: Examples of graphical representation of cosine similarity

the cosine similarity is calculated as:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\|\|\mathbf{B}\| \cos(\theta) \tag{2.7}$$

$$\text{cosine similarity}(\mathbf{A}, \mathbf{B}) = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.8}$$

where A_i and B_i refer to the values at the i-th dimension and $||\mathbf{A}||$ and $||\mathbf{B}||$ represent the norms of vector $\mathbf{A}$ and $\mathbf{B}$. θ is the angle between vector $\mathbf{A}$ and $\mathbf{B}$.

2.3 Graph Similarity Measures

In classical graph theory, a simple graph is defined as a set of vertices V and a set of edges E that connect pairs of distinct vertices (with at most one edge connecting any pair of vertices) [29]. Two graphs can have dramatically different visual outlooks even though they have the same vertices, edges, and underlying structure. This proposed the idea of graph isomorphism [30].

Definition 1. *Two graphs are isomorphic if there exists a function f from the vertices of G_1 to the vertices of G_2 (i.e., $f : V(G_1) \to V(G_2)$), such that:*

- *f is a bijection and*
- *For any two connected vertices u and v of G_1, the connectivity also exists in G_2 for $f(u)$ and $f(v)$. $E_{G1}(u, v) \implies E_{G2}(f(u), f(v))$*

The concept of graph isomorphism determines whether two graphs have the same structure. With this knowledge of isomorphism, we can intuitively gauge their similarity. A widely used metric for graph similarity is the Graph Edit Distance (GED).

Definition 2. *The GED of two graphs is defined as the minimum cost of an edit path between them, where an edit path is a sequence of edit operations (inserting, deleting, and relabeling vertices or edges) that transforms one graph into another [31].*

The costs associated with each edit operation can be customized individually for vertices and edges. Mathematically, GED can be defined as follows:

$$GED(g_1, g_2) = \min_{(e_1,\ldots,e_k)\in\mathcal{P}(g_1,g_2)} \sum_{i=1}^{k} c(e_i) \tag{2.9}$$

where $\mathcal{P}(g_1, g_2)$ represents the sequence of edit paths transforming g_1 into an isomorphic graph as g_2, and $c(e) \geq 0$ is the cost associated with each graph edit operation e. We rely on an existing Python library NetworkX[1] [32] whose GED algorithm is developed based on the work by Abu-Aisheh et al. [33].

2.4 Evaluation Metrics

The thesis incorporates two distinct evaluation processes. Initially, the proposed approach focuses on assessing a domain model against a reference domain model, which we refer to as the "**grading**" phase for clarity. Subsequently, this grading is compared to a ground truth grading assigned by a human grader. Therefore, this step evaluates the algorithm's performance by assessing the alignment between the algorithm-generated grading and human grading. We designate this phase as the "**evaluation of the algorithm**". Both evaluations employ common metrics for classification tasks such as precision, recall, and F_1-score. While the specific calculations may vary across scenarios, the fundamental principles remain constant. Precision is dependent on true positive (TP) and false positive (FP) according to

[1] https://networkx.org/documentation/stable/reference/algorithms/similarity.html

Equation 2.10, whereas recall is dependent on true positive (TP) and false negative (FN) according to Equation 2.11, and F_1-score is dependent on precision and recall according to Equation 2.12.

$$Precision = \frac{TP}{TP + FP} \tag{2.10}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.11}$$

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{2.12}$$

Since the domain model grading problem is not as simple as the binary classification task, the definitions of true positive, true negative, false positive, and false negative must be adapted accordingly as explained in more detail in Section 3.6. For the **grading** phase, the algorithm produces precision, recall, and F_1-scores for each type of model element as assessments of a domain model.

For the **evaluation of the algorithm**, more metrics are used in addition to precision, recall, and F_1-scores. The algorithm matches model elements between two domain models. To evaluate the performance of matching, we utilize precision, recall, and F_1-scores. The algorithm also generates statistical results as assessments of a domain model. We evaluate the algorithm based on how algorithm-generated results align with human-generated results. Therefore, we also employ other metrics including Kendall ranking correlation coefficient (Kendall's τ) [34] and Mean Absolute Error (MAE) (see Equation 2.13) to measure the performance of the algorithm in producing statistical results. Kendall's τ evaluates the alignment between the rankings. A value close to 1 indicates strong agreement, whereas a value close to -1 indicates strong disagreement.

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |\text{Actual}_i - \text{Predicted}_i| \tag{2.13}$$

where n is the number of samples in the dataset, $Actual_i$ represents the actual (ground truth) value for the i-th sample in the dataset, and $Predict_i$ represents the predicted value for the i-th sample in the dataset.

How to evaluate the algorithm's matching performance using precision, recall, and F_1-score will be presented in Section 4.1 and Section 5.2. How we validate the statistical results from the algorithm using MAE and Kendall's τ will be presented in Section 4.2 and Section 5.3.

2.5 Summary

In summary, this chapter describes the essential concepts and techniques that form the bedrock for the proposed algorithm in automating the assessment of domain models. Specifically, Section 2.1 drafts the textual domain model representation in EBNF format which will be input to the algorithm. Section 2.2 and Section 2.3 describe embedding techniques and graph comparisons which are fundamental to the algorithm in matching model elements. Furthermore, Section 2.4 illuminates metrics including precision, recall, and F_1-scores in the grading phase and metrics including precision, recall, F_1-scores, Kendall's τ, and MAE in the evaluation of the algorithm.

The forthcoming chapter provides an in-depth exploration of the proposed algorithm, offering detailed insights into the integration of these foundational concepts and techniques. Through this exploration, we aim to illustrate how each element contributes to the effectiveness and innovation of the automated domain model assessment approach.

Chapter 3

Method

This chapter introduces the proposed algorithm designed for the automated assessment of a domain model against a reference domain model backed by model element matching. The chapter begins with an overview of the algorithm in Section 3.1, then continues with a discussion of pre-processing the algorithm's input (see Section 3.2). Following this, Section 3.3, Section 3.4, and Section 3.5 explain the process of how the algorithm matches each type of model element between two models: classes, attributes, and relationships, respectively. Eventually, Section 3.6 and Section 3.7 illustrate the statistical outcomes of the algorithm.

3.1 Overview

The algorithm overview is shown in Figure 3.1. It consists of four main stages, *Stage 1* for *class matching*, *Stage 2* for *attribute matching*, *Stage 3* for *relationship matching*, and *Stage 4* for computing primary statistical *results*. There is also a *Stage Pre-processing* before entering the main stages (i.e., before *Stage 1* Class Matching) and a *Stage Grading* after exiting the main stages (i.e., after *Stage 4*). The algorithm takes two domain models as input, one candidate model and one reference model, and produces nested hash maps containing element matches for calculating precision, recall, and F_1-score of each kind of model element (classes, attributes, and relationships). In general, each stage matches model elements between the candidate model and reference model based on graph similarity measures (refer

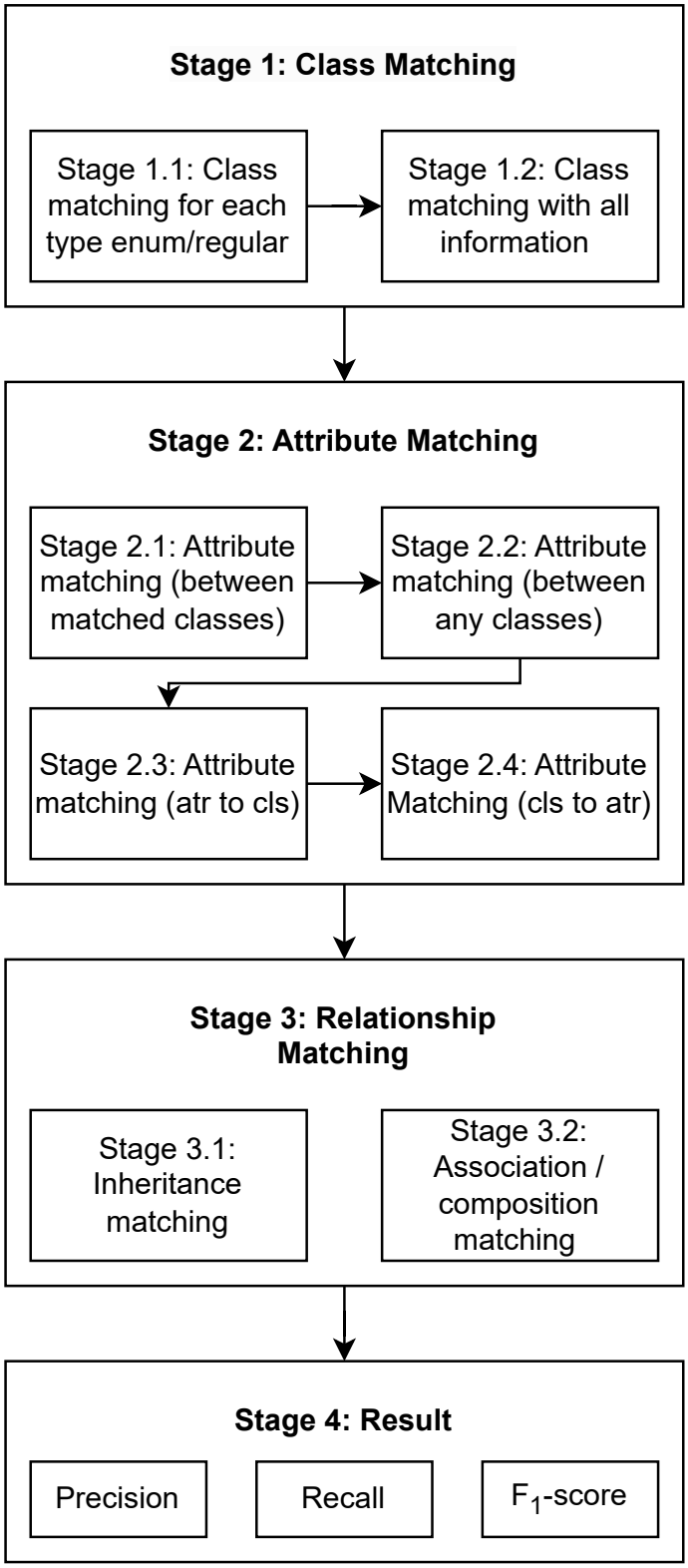


Figure 3.1: Overview of the proposed algorithm

to Section 2.3). Nested hash maps are introduced to keep track of the status of each model element from the two models, respectively, including their matching information. The hash maps also contain scores for each model element, which will be used for calculating the final results and grades. For instance, a hash map for the reference model uses its class names as the keys and if there exists a suitable counterpart class from the candidate model for a particular class in the reference model, the counterpart class will be added to this hash map and the scores will be updated accordingly.

At the pre-processing stage (*Stage Pre-processing*), the algorithm takes two domain models represented in the EBNF format (refer to Section 2.1.2) and transforms them into the desired data structures, including hash maps and lists. This preparatory step lays the foundation for subsequent stages of the algorithm by organizing the model data into accessible and manipulable formats. Then the algorithm enters *Stage 1* where it tries to match classes from the candidate model with classes from the reference model based on graph similarity measures. Once a match is found, their matching scores are determined based on pre-defined rules. After that, attributes and relationships are matched in sequence and their matching scores are calculated based on the class matching. Attribute matching also relies on graph similarity measures, whereas relationship matching is purely rule-based. At this point, the basic matching process is completed. Any unmatched model elements whose counterparts are null are considered redundant in the candidate model or missing in the reference model. Eventually, hash maps with matching information are produced. The score information in them is used to calculate precision, recall, and F_1-scores for classes, attributes, and relationships separately. The final grade is represented as a weighted average of F_1-scores.

3.2 Pre-processing

The pre-processing stage starts with initializing all the data structures needed for model comparison. To better encapsulate all the information, a model is introduced as shown in Figure 3.2. The `Grader` class has two different associations with the `Model` class. One

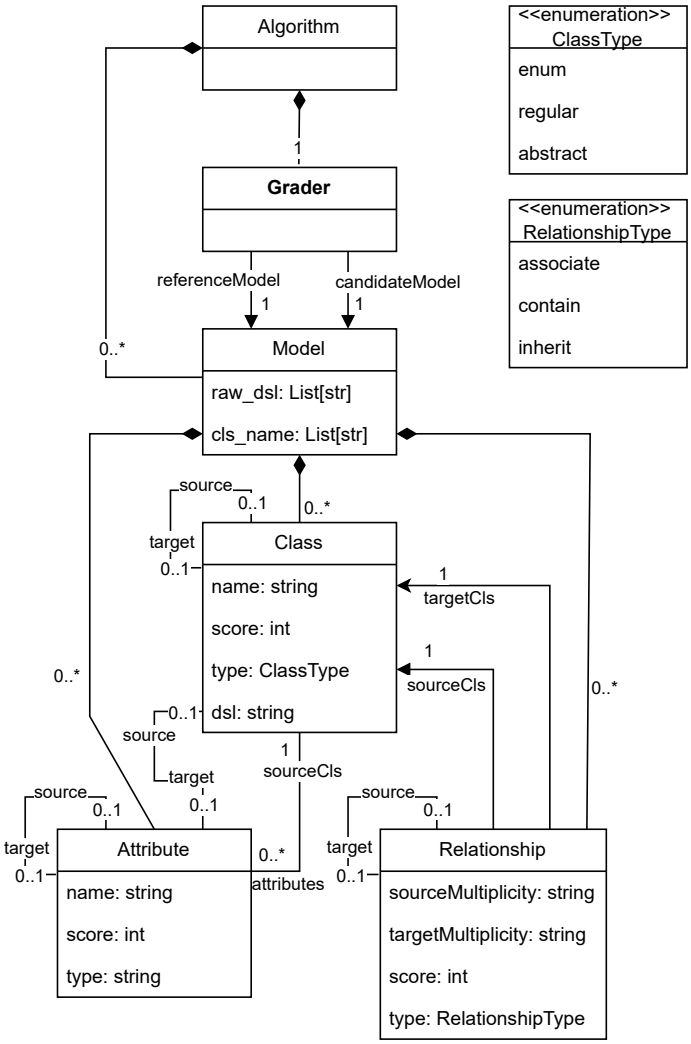


Figure 3.2: Overview of the data structure

association navigates to the reference model and the other navigates to the candidate model. The `Model` class stores information on a domain model and it has two attributes, including a list of strings storing each line of the textual domain model representation and a list of strings for class names in the model. Additionally, it possesses containment relationships with the `Class` class, `Attribute` class, and `Relationship` class. These three classes represent the three types of model elements, respectively. There exist some commonalities among them. First of all, they all have a `score` attribute, indicating the matching score. Additionally, each class incorporates a self-association to denote potential matches within its category. A match can be found between a class and an attribute. Therefore, there is an association

Table 3.1: Example of domain model representation

Domain model in EBNF format
Enumerations: DeviceStatus(Activated, Deactivated) CommandType(lockDoor, turnOnHeating) ... Classes: User(string name) Address(string city, string postalCode, string street, string aptNumber) abstract Device(DeviceStatus deviceStatus, int deviceID) ... Relationships: 1 SHAS contain * SmartHome * SensorReading associate 1 SensorDevice SensorDevice inherit 1 Device ...

between the `Class` class and the `Attribute` class. Furthermore, all three classes possess a `type` attribute, although the data type associated with this attribute varies. In the case of the `Class` and `Relationship` classes, enumeration classes define the possible values for their `type`. The `Attribute` class utilizes a string for the `type` attribute because there are too many possible data types for the `Attribute` class, e.g., int, date, float, string, etc. On the other hand, the `Class` class employs a string attribute named `dsl`, containing the EBNF-formatted textual representation of a class from the domain model. The `Relationship` class has additional attributes `sourceMultiplicity` and `targetMultiplicity`, providing further information about the multiplicities of the source class and target class. The source class and target class can be navigated from associations to `Class` class.

The algorithm reads two domain models in the EBNF format stored in the text files and pre-processes the textual representation of the domain models into the desired data structure for subsequent element matching. Each class in the textual EBNF format is expressed in one line and can be easily parsed to separate class names and attributes based on special characters. An example is shown in Table 3.1. In practice, the implementation of the `Model`, `Attribute`, and `Relationship` classes is achieved by the Python native `Dictionary` class. The concept of `Dictionary` is also known as hash map. To avoid confusion, this thesis will

use the term hash map throughout this thesis. Since class names are unique within a domain model and attribute names are unique within one class, class names can be used as the first level keys in the hash map `classes_attributes`.

```
# Hash map for classes_attributes
{
    "DeviceStatus": {
        "score": 0,
        "type": "enum",
        "dsl": "DeviceStatus(Activated, Deactivated)",
        "counterpart": None,
        "attributes": {
            "Activated": {"score": 0, "counterpart": None},
            "Deactivated": {"score": 0, "counterpart": None},
        },
    },
    ...
    "abstract Device": {
        "score": 1,
        "type": "abstract",
        "dsl": "abstract Device(DeviceStatus deviceStatus, int deviceID)",
        "counterpart": None,
        "attributes": {
            "DeviceStatus deviceStatus": {"score": 0, "counterpart": None
    },
            "int deviceID": {"score": 0, "counterpart": None},
        },
    },
    ...
}
```

Listing 3.1: Snippet of the hash map of classes

An example of the hash map is shown in Listing 3.1. This example presents relevant information for the DeviceStatus enumeration class and the abstract Device class. Meanwhile, since relationship names are not specified in the EBNF format, there could be duplicated relationships in one model. Therefore, each relationship is stored in one individual hash map, and all the hash maps for the relationships are kept in the list `relationships` as shown in Listing 3.2. The scores and counterparts for all hash maps are initialized with 0 and `None`, respectively, which will be updated once a match is found. The above-mentioned procedure of initializing hash maps is executed twice: once for the candidate model and once for the reference model.

```
# List of hash maps for relationships
[
    {
        "dsl": "1  SmartHomeApplicationSystem contain  * SmartHome",
        "score": 0,
        "counterpart": None,
    },
    {
        "dsl": "1 SensorDevice associate  * SensorReading",
        "score": 0,
        "counterpart": None,
    },
    {
        "dsl": "SensorDevice inherit  Device",
        "score": 0,
        "counterpart": None,
    },
    ...
]
```

Listing 3.2: Snippet of the list for relationships

3.3 Stage 1: Class Matching

In the first stage of the model assessment process, the algorithm focuses on matching classes between the candidate model and the reference model (referred to as candidate classes and reference classes for simplicity). This process starts with calculating the class embeddings. The embeddings of a class are derived from the embeddings of its class name, attributes, and relationships. The next step is calculating pair-wise cosine similarities between classes based on the class embeddings. Then the algorithm matches classes by deriving a list of operations to bring the candidate model to the reference model. In practice, the algorithm converts domain models into two graphs. Then it matches vertices (represents classes in the corresponding domain model) by the candidate graph isomorphic to the reference graph where the cost of each operation is derived from the similarity score between the matched graph elements.

This stage can be divided into two smaller stages: matching classes of the same type and matching classes of any type with all available information.

3.3.1 Stage 1.1: Class Matching within Types

In *Stage 1.1*, the algorithm focuses on matching classes based on names and attributes within the same type. There are three types of classes: regular classes, abstract classes, and enumeration classes. Regular classes and abstract classes share numerous commonalities, whereas enumeration classes demonstrate differences between the others. For instance, both regular classes and abstract classes can have relationships with other classes, but this does not happen in enumeration classes. Meanwhile, it is often the case that there are only a very limited number of abstract classes presented in the domain model. Therefore, in this sub-stage, the algorithm groups regular classes and abstract classes into one matching pool and treat enumeration classes in another matching pool. In other words, the algorithm matches enumeration classes to enumeration classes and non-enumeration to non-enumeration classes.

```
# Input: reference classes in reference_classes
#        candidate classes in candidate_classes
# Return: Enumeration class match in enum_mapping
#         Non-enumeration class match in cls_mapping
similarities = dict()
for ref_cls in reference_classes:
    E_c_r = E_c(ref_cls.name) # Class-name-only embedding for reference class. See Equation 3.4
    E_c_A_r = E_c_A(ref_cls.name, ref_cls.attributes)  # Class-name-with-attributes embeddings. See Equation 3.5

    for can_cls in candidate_classes:
        E_c_c = E_c(can_cls.name) # See Equation 3.4
        E_c_A_c = E_c_A(can_cls.name, can_cls.attributes)  # See Equation 3.5
        s_name = cosine_similarity(E_c_r, E_c_c) # similarity from class-name-only embeddings
        s_attribute = cosine_similarity(E_c_A_r, E_c_A_c) # similarity from class-name-with-attributes embeddings
        similarity = percentage * s_name + (1-percentage) * s_attribute
        similarities[reference_cls][candidate_cls] = similarity

enum_mapping = match_classes(reference_enumeration_classes, candidate_enumeration_classes, ...)
cls_mapping = match_classes(reference_non_enumeration_classes, candidate_non_enumeration_classes,...)
```

Listing 3.3: Pseudo-code for class matching process

Listing 3.3 demonstrates the pseudo-code for the class matching process. The algorithm starts with utilizing word embeddings to convert class names and attributes into vector space. Specifically, each class name or attribute name is converted into an N-dimensional vector using Equation 3.4 and Equation 3.5, respectively. Then the algorithm averages all the embeddings to produce the class embeddings. Subsequently, pair-wise similarities between classes can be calculated using cosine similarity discussed in Section 2.2.3. With pair-wise cosine similarities, the algorithm matches classes using the `match_classes()` function.

We want to match classes of the same type. Therefore, candidate classes and reference classes are divided into four lists:

- `reference_enumeration_classes`
- `reference_non_enumeration_classes` (regular and abstract)
- `candidate_enumeration_classes`
- `candidate_non_enumeration_classes` (regular and abstract)

and they will be passed into the **`match_classes()`** function to compute the class matches.

Embedding Approaches. Embedding a class name or attribute name sometimes can be cumbersome if the word is not recognizable by the word embedding method. For example, a class name may consist of several words in camel case and there could be multiple attributes for a class. The algorithm addresses words in the camel case by dividing them into individual words and taking average embeddings of each word to represent this model element. Let us define a function for calculating word embeddings:

$$E = \texttt{word\_embedding}(\text{word}) \tag{3.1}$$

where $E \in \mathbb{R}^d$. d is the dimension of the word embedding (refer to Section 2.2). We have a function called `split_camel_case()`, which takes a model element e in camel case and splits

it into a list of individual words W where W_i represents the i-th word in the list.

$$W = \texttt{split\_camel\_case}(e) \tag{3.2}$$

Then the embedding of e is defined as the element-wise average embedding of words in W using the `get_embedding` function:

$$\texttt{get\_embedding}(e) = \frac{\sum_{i=1}^{n} \texttt{word\_embedding}(W_i)}{n} \tag{3.3}$$

where n is the length of the list of split words W and W_i is an individual word among split words. The same strategy is applied to embed each attribute as well, which is to split the attribute name first, get split words embedded separately, and then find the average. Furthermore, it is possible that a special class name or attribute name might not be recognized by the employed word embedding technique. For instance, this could occur with composite names formed by concatenating the initial letters of multiple words. In such cases, the algorithm uses the average of embeddings of each letter in the name.

There are two types of embedding to be calculated.

- Class-name-only embeddings
- Class-name-with-attributes embeddings

For the class-name-only embeddings, since each class only has one name, the algorithm can use the above-mentioned embedding approaches to embed the class name into a vector and use the vector as the embedding representation of the class. Let us refer to a class name as c.

$$E_c = \texttt{get\_embedding}(c) \tag{3.4}$$

For the second type of embedding, the algorithm needs to take class names and attributes into account. Therefore, after calculating vectors representing the class name and each attribute separately, the algorithm takes the average of all the vectors as the final vector

representation of the class. Let us define a list of attributes A where A_i represents the i-th attribute for a class. Let us refer to a class name as c. The class-name-with-attributes embeddings of a class can be calculated as the following:

$$E_{c,A} = \texttt{get\_embedding}(c, A) = \frac{\texttt{get\_embedding}(c) + \sum_{i=1}^{n} \texttt{get\_embedding}(A_i)}{1+n} \tag{3.5}$$

where n is the length of the list of attributes A for the class.

Cosine Similarities The algorithm calculates the class-name-only and class-name-with-attributes cosine similarities separately between each pair of classes in the reference model and candidate model based on their respective class embeddings. The cosine similarity between a pair of classes can be defined as the following:

$$c_{i,j} = \texttt{cosine\_similairy}(e_i, e_j) \tag{3.6}$$

where e_i is the embedding of i-th class c_i in the reference model and e_j is the embedding of j-th class c_j in the candidate model (refer to Section 2.2.3). Notably, two types of embedding result in two cosine similarities. A weighted average is applied to combine the similarity scores from class-name-only embeddings and class-name-with-attributes embeddings for aggregation. The weight is represented by the variable `percentage`, which can be adjusted to control the importance of each similarity:

$$c_{i,j} = \texttt{percentage} * c^{'}_{i,j} + (1 - \texttt{percentage}) * c^{''}_{i,j} \tag{3.7}$$

where $c^{'}_{i,j}$ is a similarity from class-name-only embeddings and $c^{''}_{i,j}$ is a similarity from class-name-with-attributes embeddings. Eventually, all the cosine similarities are stored in a hash map where the first-level keys are reference class names, the second-level keys are candidate class names, and values are cosine similarities. It can be represented as the following:

$$\texttt{similarities}[c_i][c_j] = v_{i,j} \tag{3.8}$$

where c_i is a reference class; c_j is a candidate class; and $v_{i,j}$ is the aggregated similarity.

One more data processing task is finding the exact match between classes. The algorithm iterates through all the cosine similarities and looks for similarity values larger than 0.99. A class pair with a similarity value larger than 0.99 is considered an exact match. If such an exact match is found between a reference class and a candidate class, the cosine similarities between this reference class and all the other candidate classes are set to 0. Practically, this data processing task means exact match class pairs will not be considered for any other matching.

`match_classes()` Function. In the next step, the `match_classes` function matches classes with graph edit distance and similarities. This function requires five inputs: a list of reference class names, a list of candidate class names, a hash map where cosine similarities are stored, a float parameter named `threshold`, and a boolean parameter named `verbose`. Inside this function, two graphs are initialized. One contains vertices representing reference classes and the other contains vertices representing candidate classes. This function finds the optimal graph edit distance (refer to Section 2.3) between the two graphs. Since both graphs only contain vertices, possible graph edit operations include vertex insertion, deletion, and substitution. The cost of insertion and deletion is set to one by default, whereas the cost of substitution is defined as *1 - cosine similarity*. The intuition behind this is to match vertices with high similarities. The optimal graph edit distance includes a list of operations with the lowest cost. Therefore, high-similarity vertex pairs have low cost and they are more likely to get matched. Since the algorithm wants to filter some class pairs whose similarities are low, a threshold parameter is utilized. If the similarity is lower than the threshold, the cost of substitution will be adjusted to 3, which is larger than the total cost of one deletion and one insertion combined (cost of 2). This prevents substituting low-similarity class pairs and instead performs one deletion and one insertion. Overall, this function optimizes the sequence of vertex edit operations transforming one graph to the other graph with the lowest cost. It returns a list of tuples containing class matches including None-match. Each tuple

contains two elements: one reference class and one candidate class. Only one of them can be `None`. One example is shown in Listing 3.4. DeviceStatus class is matched with DeviceType, whereas RuleStatus is matched with `None`.

```
[("DeviceStatus", "DeviceType"),
("CommandType", "CommandType"),
("CommandStatus", None),
("RuleStatus", None),
("BinaryOp", "Operator")]
```

Listing 3.4: Example of class matches

3.3.2 Stage 1.2: Class Matching with All Information

Stage 1.2 involves matching leftover classes in the candidate model and the reference model from the previous stage using all available information, including attributes and relationships. A leftover class is one that lacks a match. This can occur when a candidate class has no corresponding reference class, or when a reference class has no corresponding candidate class. For each class, the algorithm gathers its raw class representation in EBNF representation, along with all associated relationships. This information is stored in the string format. Subsequently, the algorithm employs sentence embedding techniques (refer to Section 2.2.2) to transform the aforementioned strings into vector representations, thus representing each class in vector space. Next, the algorithm uses the class embeddings to calculate the pair-wise cosine similarities between the reference classes and the candidate classes. This class matching process is similar to Listing 3.3 with differences in the class embedding part.

To match leftover classes, the `match_classes()` function is used again, but with different inputs. The inputs now include a list of leftover reference classes, a list of leftover candidate classes, and a hash map containing their pair-wise cosine similarities based on sentence embeddings. A threshold parameter is also utilized to prevent matching low-similarity class pairs. This function matches leftover classes and returns class matches in a list of tuples.

Score Assignment. The algorithm assigns scores to each element in the matches. For both *Stage 1.1* and *Stage 1.2*, after finding class matches with the `match_classes` function,

the algorithm checks whether class pairs possess the same type (enumeration, regular, or abstract). If both classes in a class pair have the same type, they are assigned a score of one. The hash maps `classes_attributes` are updated where the scores are changed to one and the counterparts are also updated accordingly. If two classes in a pair have different class types, the score is changed to 0.5.

3.4 Stage 2: Attribute Matching

In *Stage 2*, the algorithm focuses on matching the attributes between the reference and candidate models (referred to as reference attributes and candidate attributes for simplicity). Leftover classes from previous stages are also considered with a low priority. This stage is further divided into four sub-stages:

- *Stage 2.1* Attribute matching between matched classes
- *Stage 2.2* Attribute matching between any classes
- *Stage 2.3* Reference attribute to candidate class matching
- *Stage 2.4* Reference class to candidate attribute matching

In general, attribute matching follows a similar logic to class matching, which is based on embeddings, cosine similarities, and optimal graph edit distance.

3.4.1 Stage 2.1: Attribute Matching Between Matched Classes

In this stage, reference attributes and candidate attributes that are in the matched classes are considered for matching. Therefore, the first step is to get all the class matches from *Stage 1*. To achieve this, the algorithm iterates through the hash map `classes_attributes` (refer to Listing 3.1) containing the matching information to find all reference classes whose counterpart is not `None`. If such a counterpart exists, it creates a pair `[reference_class, counterpart]` and appends it to the `pairs` list. After getting this list, the algorithm starts a for loop to proceed to map attributes between the class pairs.

`match_attributes` Function. For each class pair, the algorithm collects reference attributes and candidate attributes in two separate lists. These two lists are passed into the `create_cosine_similarity_dict` function along with a word embedding. Inside this function, attributes are embedded using the `get_embedding` function (refer to Equation 3.3) for computing pair-wise cosine similarities which are stored in a nested hash map. Subsequently, the two lists and the hash map are passed to the `match_attributes` function to compute attribute matching. This function is similar to the `match_classes` function. Two graphs are initialized where vertices represent reference attributes and candidate attributes, respectively. The optimal graph edit distance is computed with deletion and insertion cost to be one and substitution cost to be *1 - cosine similarity*. A threshold is set to filter out matching low-similarity pairs. The `match_attributes` function also returns a list of attribute matches in tuples.

Score Assignment. The algorithm assigns matching scores to each attribute in the returned matches. The algorithm iterates through the obtained matches and updates the attribute information for both the reference hash map and the candidate hash map named `classes_attributes`. Each attribute includes a type, e.g., int, string, or enumeration class. Sometimes there could be an anti-pattern in the type. For instance, using a regular class name as a type is a common anti-pattern. Therefore, the algorithm conducts *type checking* on matched attributes in each pair with three rules:

- **Rule 1**: If any type is a regular class or abstract class, mark them as partially correct.
- **Rule 2**: If both types are a matched enumeration class, mark them as correct. Otherwise, it is partially correct.
- **Rule 3**: If both types are matched primitive types, mark them as correct. Otherwise, they are matched as partially correct. For instance, the `int` type can be matched with a `float` type or a `double` type but cannot be matched with a `string` type.

For attribute pairs with correct type checking, their scores are assigned to one. Otherwise, their scores are assigned to 0.5. This score assignment is also used for *Stage 2.2*, *Stage 2.3*, and *Stage 2.4*.

3.4.2 Stage 2.2: Attribute Matching Between Any Classes

Some attributes can be misplaced in the incorrect classes but they can still make sense to the overall domain modeling. Therefore, this stage focuses on matching leftover attributes without restricting their source classes. The first task is getting unmatched reference attributes and candidate attributes in two lists. The algorithm wants to prioritize matching attributes whose source classes are more similar. Therefore, source classes are also collected in two lists. Next, two sets of cosine similarity scores are calculated and stored in two nested lists. One is class-to-class pair-wise cosine similarity, and the other is attribute-to-attribute cosine similarity. Both types of similarity are based on word embedding. After that, two nested lists of cosine similarities are combined into a nested hash map whose first level key is in the format of a tuple `(reference_attribute, source_class)` and the second level key is also in the format of a tuple `(candidate_attribute, source_class)` and the value is the cosine similarity. A weighted average is used in combining pair-wise attribute-to-attribute and class-to-class similarity into one value. Attributes are unique within a class. However, different classes can have the same attribute name. This is why the keys need to be combined with class names to ensure each attribute will have a unique representation even if they have the same name across different classes. Once the combined cosine similarity is obtained, they are passed to the `match_attributes` function (the same one used in the previous stage) for matching. Then the algorithm iterates through generated matches and updates the attribute-matching information. The algorithm also conducts the same type checking as described in Section 3.4.1.

3.4.3 Stage 2.3: Reference Attribute to Candidate Class Matching

One common anti-pattern in domain modeling is mistakenly having an attribute instead of a class. For instance, an address class is more suitable in some contexts rather than an address attribute. Despite being an anti-pattern, it can still bring partial value to the modeling practice. This stage is intended to identify any match between a reference attribute and a candidate class. First of all, leftover reference attributes and leftover candidate classes are collected into two separate lists. Two sets of similarity scores are created attribute-to-class and class-to-class similarities based on word embeddings. The class-to-class similarity is the same as for the previous sub-stage. The attribute-to-class similarities are calculated between the reference attributes and the candidate classes. Reference attributes and candidate classes are embedded with a word embedding approach and the algorithm calculates the pair-wise similarities between them. Then the algorithm follows the same procedure of combining two similarities lists into one hash map and then utilizes the `match_attributes` function to match attributes to classes. For each matching pair, since this is an anti-pattern, the score is set to 0.5 and the matching information is updated at the `classes_attributes` hash maps.

3.4.4 Stage 2.4: Reference Class to Candidate Attribute Matching

In *Stage 2.4*, the algorithm finds a reference class matching a candidate attribute. It follows a similar logic to *Stage 2.3* whereas now the class-to-attribute similarity is calculated between leftover reference classes and leftover attributes. The other steps are identical to *Stage 2.3*.

3.5 Stage 3: Relationship Matching

In *Stage 3*, the algorithm matches relationships based on class matches. This process is illustrated in pseudo-code in Listing 3.5. Relationship matching is rule-based, which is different from class or attribute matching. The algorithm utilizes nested loops to iterate through all combinations of relationships from the candidate model and the reference model. For each relationship in the reference model, a list called `matchings` is initialized. Then, for

each relationship in the candidate model, if both relationships are unmatched, a potential match is examined using the `compare_edges` function. The matching is appended to the list of `matchings`. After comparing all possible combinations, the best match among the matches is found. The algorithm keeps track of the matching status using a list of hash maps for relationships (refer to Listing 3.2). This ensures that relationships that have already been matched are not compared again.

```
for reference_rel in reference_rels:
  matchings = set()
  for candidate_rel in candidate_rels:

    if (reference_rel, candidate_rel) not in matchings:
      matching = compare_edges(reference_rel, candidate_rel)
      matchings.append(matching)

  Find the matching with the highest score among matchings
  update the hash map for reference_rel
  update the hash map for candidate_rel
```

Listing 3.5: Pseudo-code for relationship matching

`compare_edges()` Function. *Stage 3* relies on the `compare_edges()` function to examine whether two relationships can be matched. This function takes two relationships in string format as input and tries to find an exact match, a partial match, or no match. There are three kinds of relationships defined in the EBNF format: `inherit`, `contain`, and `associate`. An `inherit` relationship only consist of three elements: child class, inherit, and parent class, e.g., SensorDevice inherit * Device. On the other hand, both `associate` and `contain` relationships consist of five elements: multiplicity 1, class 1, relationship type, multiplicity 2, and class 2, e.g., 1 Room contain * SensorDevice. Therefore, the input relationships are classified based on the difference in the number of elements at first. If both relationships contain three elements, they enter *Stage 3.1* Inheritance matching. In this stage, the algorithm checks whether the second element is `inherit`. Meanwhile, the algorithm checks if the first elements (child classes) in both relationships are the counterparts of each other. In other words, the classes are matched from previous stages. The third elements (parent

classes) are also checked to confirm whether they are counterparts of each other. If all three conditions are satisfied, an exact match between these two relationships is established.

On the other hand, if both relationships contain five elements, they enter *Stage 3.2* Association/composition matching. An exact match is found if and only if all the five elements match each other. Two relationships have the same type. Multiplicities are equal, and classes are counterparts of each other. Notably, for `associate` relationships, a situation that should considered is that the position of classes and multiplicities can be flipped. For instance, 1 SensorDevice associate * SensorReading is identical to * SensorReading associate 1 SensorDevice. Additionally, partial matches are considered. If the above exact match fails, the function ignores multiplicities and types but only checks whether classes are counterparts of each other. For example, 1 Room contain * Device and * Room associate * Device make up a partial match.

The `compare_edges` function eventually returns a boolean and one score. The boolean indicates the matching status, while the two scores represent the matching scores for the input relationships. If two relationships are matched or partially matched, a boolean value of `True` is returned. The score is the matching score for both the input reference relationship (denoted as `reference_rel` in Listing 3.5) and for the input candidate relationship (denoted as `candidate_rel` in Listing 3.5). It returns `(True, 1)` for exact matches, `(True, 0.5)` for partial match, and `(False, 0)` for no match. If a reference relationship is matched with multiple candidate relationships, the algorithm finds the match with the highest matching score and ignores other matches. After that, the scores and counterparts are then stored in the hash map for each relationship based on the best match.

3.6 Stage 4: Result

In *Stage 4*, the algorithm calculates precision, recall, and F_1-scores as the evaluation of the domain model. In general, the above stages match each model element and assign it a score from the set $\{0, 0.5, 1\}$. All the matching information is stored in the designed data structure

(See Listing 3.1 and Listing 3.2) with a score for each model element. The scoring function S for an element x can be expressed as follows:

$$S(x) = \begin{cases} 1 & \text{if } x \text{ is in a perfect match} \\ 0.5 & \text{if } x \text{ is in a partial match} \\ 0 & \text{if } x \text{ is in no match} \end{cases} \tag{3.9}$$

To generate a comprehensive assessment of a domain model, the algorithm uses a modified version of the precision, recall, and F_1-scores metrics over the classes, attributes, and relationships. Precision measures the overall correctness of model elements in the candidate model. For example, let $\mathcal{C}$ be the set of all classes and enumerations in the candidate model of size $m = |\mathcal{C}|$, the precision of classes in the candidate model can be expressed as

$$Precision_{\mathcal{C}} = \frac{\sum_{i=0}^{m} S(\mathcal{C}_i)}{m}, \tag{3.10}$$

where S is the scoring function defined in Equation 3.9.

Recall measures the degree of the reference model covered by the candidate model. Let $\mathcal{C}_{ref}$ be the set of classes and enumerations in the *reference model* (ground truth) of size $n = |\mathcal{C}_{ref}|$, the recall of classes can be expressed as

$$Recall_{\mathcal{C}} = \frac{\sum_{i=0}^{n} S(\mathcal{C}_i)}{n}. \tag{3.11}$$

Finally, the algorithm uses the classical F_1-score definition:

$$F_1^{\mathcal{C}} = \frac{2 \times Precision_{\mathcal{C}} \times Recall_{\mathcal{C}}}{Precision_{\mathcal{C}} + Recall_{\mathcal{C}}}. \tag{3.12}$$

Metrics for attributes or relationships can be computed by substituting $\mathcal{C}$ with the set of attributes or relationships.

3.7 Grading

The algorithm needs to return a final grade for an evaluated domain model. F_1-scores are calculated for each model element type, and the algorithm uses the weighted average of F_1-scores to produce the final grade of this domain model. The weights can be easily adjusted to adapt to different real-world grading schemes using the following formula:

$$grade = \frac{w_c}{w_c + w_a + w_r} F_1^{\mathcal{C}} + \frac{w_a}{w_c + w_a + w_r} F_1^{\mathcal{A}} + \frac{w_r}{w_c + w_a + w_r} F_1^{\mathcal{R}} \tag{3.13}$$

where w_c, w_a, and w_r are weights for classes, attributes, and relationships, respectively. $F_1^{\mathcal{C}}$, $F_1^{\mathcal{A}}$, and $F_1^{\mathcal{R}}$ are the F_1-scores for classes, attributes, and relationships, respectively.

Based on the experiment in assessing students' submissions in university-level modeling practice, a class typically weighs four points whereas an attribute or a relationship weighs one point each. Motivated by this observation, the final grade of a domain model is calculated with $w_c = 4$, $w_a = 1$, and $w_r = 1$.

3.8 Summary

In conclusion, this chapter explains how the proposed algorithm assesses a domain model against a reference model based on model element matching. Section 3.1 gives a high-level overview of the algorithm and briefly introduces different stages of the algorithm. It also mentions the core logic of the algorithm, which is matching model elements between the candidate model and reference model based on graph similarity measures. Section 3.2 explains the data structure needed for information storage through the algorithm. A model is proposed for encapsulating all the information with a practical implementation using the Python native `Dictionary` class. This section also discusses how to process the textual domain model input into the designed data format. Then Section 3.3 explains how the algorithm matches classes between the candidate model and the reference model. The algorithm embeds classes into vectors and then finds pair-wise cosine similarity scores between classes.

With cosine similarity scores, the algorithm matches classes based on graph similarity, i.e., a GED algorithm. Section 3.5 talks about how the algorithm matches attributes between the two models. The process of attribute matching employs a logic similar to that used in class matching with minor differences. The calculation of the pair-wise cosine similarity score between attributes is slightly different because the algorithm needs to consider the source class of an attribute in certain sub-stages. Section 3.5 employs a rule-based matching process, which is different from those used in class matching or attribute matching. Concrete rules are defined to determine whether two relationships are in an exact match, partial match, or no match. This stage does not involve embeddings or graph similarity measures. Subsequently, Section 3.6 discusses how to use the matching results to calculate precision, recall, and F_1-scores for each type of model element as the evaluation of the domain model. Eventually Section 3.7 discusses how to use F_1-scores for classes, attributes, and relationships to produce an aggregated final grade. A generalized formula is proposed. The next chapter provides insights into how the algorithm is evaluated.

Chapter 4

Algorithm Evaluation

This chapter covers how to evaluate the algorithm with an automation pipeline to compare matches. This thesis focuses on automated domain model assessment, in which a clear reference model and a candidate model are given. The essence of the proposed algorithm is to match model elements in the candidate model to those in the reference model. Therefore, an ideal algorithm should match all the model elements perfectly and align with matches made by a human grader. Therefore, Section 4.1 demonstrates how the algorithm is evaluated in terms of the matches it produced against the matches generated by the author.

Moreover, the algorithm generates statistical results for the candidate model based on the matched matches. A domain model can be assessed with different metrics. For instance, in our previous work [2], we manually evaluated LLM-generated domain models with precisions, recalls, and F_1-scores to analyze the modeling ability of LLMs. For modeling practices in undergraduate-level, model-driven programming courses (e.g., exams, projects), it is more typical to assign numerical grades or letter grades to domain models submitted by students. The proposed algorithm assesses a domain model against a reference model in precisions, recalls, and F_1-scores. Additionally, an aggregated grade is also produced. Therefore, Section 4.2 presents how the algorithm is evaluated in terms of the statistical results it produced against the same set of statistical results produced by human graders.

Table 4.1: Scheme for comparing two domain models [2]

Category	Description	Examples
c1	Direct match	H2S():H2S(), Person(...):User(...)
c2	Semantically equivalent	SecondHandArticle(boolean discarded,...): SecondHandArticle(Status status,...), Status(AVAILABLE, DISCARDED)
c3	Partial match	0..1 Route associate * Item: (1) 1 Route associate * Item (2) 0..1 Route associate * FoodItem
c4	No match	abstract UserRole():(No role class)

4.1 Evaluation of Generated Matches

This section explains how to evaluate the algorithm by comparing matches the algorithm produced with matches made by a human grader.

4.1.1 Human Matches

Element matches are made manually by the author of this thesis. The manual matching follows the same procedure as proposed in our last paper [2]. Each model element is matched into one of four categories as shown in Table 4.1.

c1 Category *c1* includes all domain elements that have an exact match with the elements of the same type in the reference model. It is important to note that this match is based on semantics rather than string comparison.

c2 Elements in category *c2* are matched with an element that may not be the same type in the reference model but are semantically equivalent to the matched element.

c3 Category *c3* is for elements that only partially match the element in the reference model. This category also includes elements without a match in the reference model due to another incorrect design decision (i.e., consequential mistakes) that are otherwise correct.

c4 Category *c4* captures incorrect elements that do not match any elements in the reference model.

All elements are matched in *both* the candidate and reference model to compute the precision and recall. Each element is assigned a score. Specifically, elements in the first two categories are awarded 1 point, while elements in *c3* receive 0.5 points, and those in *c4* receive 0 points. The scoring function S for an element x can be expressed formally as follows:

$$S(x) = \begin{cases} 1 & \text{if } x \text{ is in c1 or c2} \\ 0.5 & \text{if } x \text{ is in c3} \\ 0 & \text{if } x \text{ is in c4} \end{cases} \tag{4.1}$$

Human-generated matches are also kept in the same data structure (see Section 3.2) for automatizing the evaluation process. Once the two collections of element matches are prepared, the algorithm's matching performance can be evaluated using precision, recall, and F_1-scores.

4.1.2 Comparison of Matches

The general workflow of evaluating matches is shown in Figure 4.1. For one assessment of a domain model, there is a collection of matches produced by the proposed algorithm. There is another collection of matches produced by a human grader following the procedure in Section 4.1.1. Let us define two collections of matches as M_A (algorithm matches) and M_H (human matches), where $M_A = \{m_{a1}, m_{a2}...m_{an}\}$ and $M_H = \{m_{h1}, m_{h2}...m_{hn}\}$. Then let us explain the component of an individual match: m_i. All the matching information is stored in the hash maps as mentioned in Section 3.2. To remove unnecessary information, the pipeline extracts matches from the hash maps and reformats each match into the form of a tuple data structure: `(element_name, counterpart, matching_score)`. This is a basic form of m_i. Notably, the algorithm assigns a score of $\{0, 0.5, 1\}$ to each element in a match.

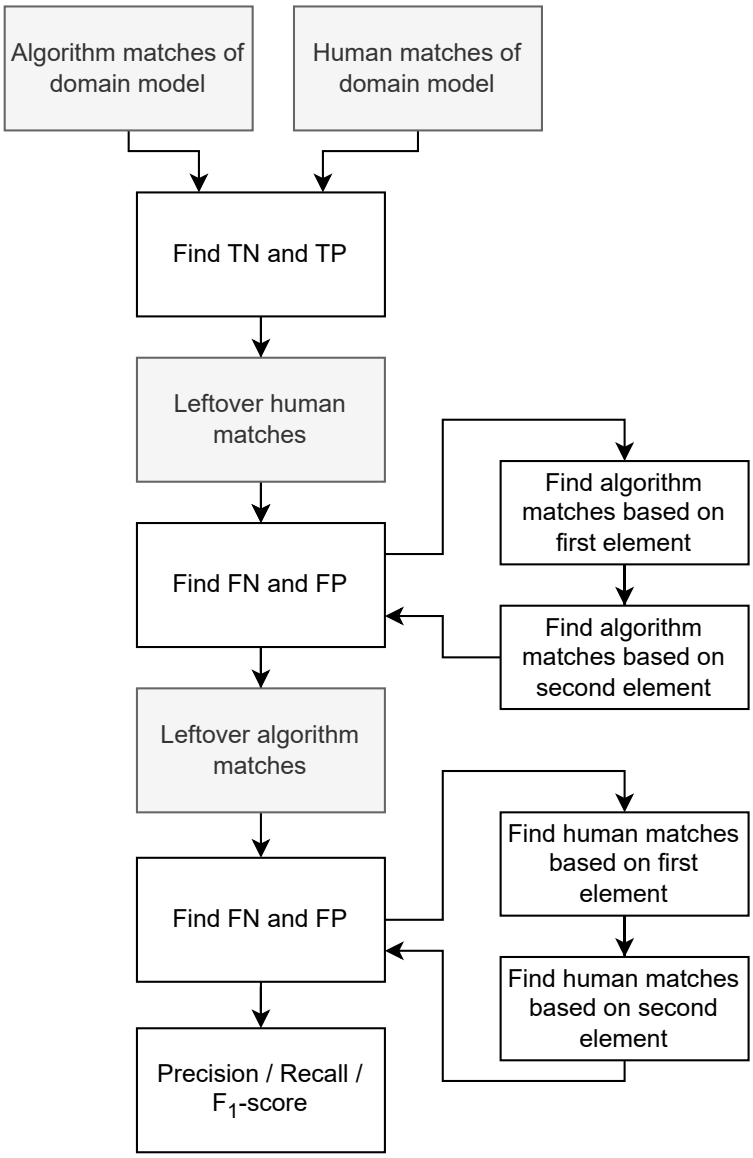


Figure 4.1: Workflow of evaluating matches

This is the `matching_score` in the tuple. To keep matches consistent, the first element in the tuple is always an element from the reference model or `None`.

Specifically, the pipeline first iterates through reference model elements. For each element, a tuple is created in the form of `(element_name, counterpart, matching_score)`. Then the pipeline iterates through candidate model elements and creates matching tuples in the form of `(counterpart, element_name, matching_score)` so that the first element in the tuple is either from the reference model or `None`. To prevent double counting, any matching tuple created during iterating candidate model elements will be checked for duplication before appending to the list M_A or M_H. M_A and M_H are shown in the top part of Figure 4.1: *Algorithm matches of domain model* and *Human matches of domain model*. The following paragraphs will explain how to compare these two collections of matches in the context of domain model evaluation.

Labeling Strategy

After getting two collections of matches, the pipeline can proceed to compare them by classifying them into: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). The pipeline only focuses on the first two elements in the match in *Matching Comparison*, while considering the matching score in *Score Assignment*.

Match Comparison. Let a match m_i contain two elements (m_i^1, m_i^2), where the first element, m_i^1, represents a model element from the reference model or None, and the second element, m_i^2, represents a model element from the candidate model or `None`. Only one of them can be `None`. Assume a match from M_A as m_{ai} and a match from M_H as m_{hi}. Then all the situations of m_{ai} and m_{hi} pairs and their corresponding evaluation labels are demonstrated in Table 4.2. For example, consider a reference class SensorDevice, a candidate class Sensor, another candidate class ActuatorDevice, and a correct human match (SensorDevice, Sensor). If the algorithm match is (SensorDevice, Sensor), this represents a TP in situation (1). If the algorithm match is (SensorDevice, None), this represents an incorrect match pair in

Table 4.2: Different scenarios of evaluating matches, where a, b, and c represent model elements

Situation	Human Matches		Algorithm Matches		Score	Label
	reference	candidate	reference	candidate		
1	a	b	a	b	1	TP
2	a	b	a	None	0	FN
3	a	b	a	c	0	FP
4	a	None	a	b	0	FP
5	a	None	a	None	1	TN
6	None	a	b	a	0	FP
7	None	a	None	a	1	TN
8	b	a	None	a	0	FN
9	c	a	b	a	0	FP

situation (2). If the algorithm matches (SensorDevice, ActuatorDevice), this represents an incorrect match pair in situation (3). It should be noted that, at this point, there are no match pairs that receive partial scores. Each match pair either scores 0 or 1. Partial scores are considered later when taking the matching score of each match into account. The details of each situation and its label are explained below.

1. (a, b) vs. (a, b). The human grader matches element `a` with element `b`, and the algorithm also matches element `a` with element `b`. This is considered as a true positive TP with a score of 1.

2. (a, b) vs. $(a, None)$. The algorithm matches element `a` with nothing (i.e., `None`). However, `a` is supposed to be matched with something (i.e., `b`). Therefore, this is considered as a false negative FN with a score of 0.

3. (a, b) vs. (a, c). The algorithm matches element `a` with `c`. However, `a` is supposed to be matched with another element `b`. Therefore, this is considered as a false positive FP with a score of 0.

4. $(a, None)$ vs. (a, b). The algorithm matches an unnecessary element to `a`. Therefore, this is a false positive (FP) with a score of 0.

5. $(a, None)$ vs. $(a, None)$. Both the human grader and the algorithm also match element `a` with nothing. Therefore, this is considered as a true negative TN with a score of 1.

6. $(None, a)$ vs. (b, a). Assume the second element is the focus point. This situation is the same as the fourth situation with an FP and a score of 0.

7. $(None, a)$ vs. $(None, a)$. This situation is the same as the fifth situation with a TN and score of 1.

8. (b, a) vs. $(None, a)$. This situation is the same as the second situation with an FN and a score of 0.

9. (c, a) vs. (b, a). This situation is the same as the third situation with an FP and a score of 0.

Score Assignment. For match pairs labeled as `TP` or `TN`, the pipeline further examines whether the third element `matching_score` is consistent in both matches. If they are consistent, the pipeline assigns a score of 1 to the match pair. If not, the pipeline assigns a score of 0.5 to the match pair. In other words, `TP` and `TN` are sometimes only worth 0.5 instead of 1. Meanwhile, the pipeline marks these match pairs as evaluated. Subsequently, the pipeline obtains leftover M_A and leftover M_H whose m_i have not been evaluated.

Evaluation Workflow

Using the labeling strategy, the pipeline proceeds to the step of *Find TN and TP* as shown in Figure 4.1. This step is finding the intersection of M_A and M_H with the comparison rule defined as an m_{ai} is considered to be identical to an m_{hi} if m_{ai}^1 (the first element) in m_{ai} is identical to m_{hi}^1 (the first element) in m_{hi} and m_{ai}^2 (the second element) in m_{ai} is also identical to m_{hi}^2 (the second element) in m_{hi}. After finding a pair of identical matches, the pipeline needs to determine if it is `TP` or `TN`. If there exists `None` in this pair of matches, then this pair is considered as `TN`. Otherwise, this pair is `TP`.

After the previous step, the pipeline focuses on the *Leftover human matches* and proceeds to *Find FN and FP* in the leftover M_H (see Figure 4.1). This step can be further divided into two small steps. (1) For each leftover human match m_{hi} with elements (a, b), the pipeline can try to find an algorithm match m_{aj} with elements (a, x) where $x \neq b$. This is finding an algorithm match based on the first element. (2) If the pipeline cannot find such a m_{aj}, it shifts the focus to the second element and tries to find an algorithm match m_{aj} with elements (x, b) where $x \neq a$. After finding such a match pair, the pipeline classifies them into `FP` or `FN`.

After the previous step, there may still exist some leftover algorithm matches. The pipeline focuses on the *Leftover algorithm matches* (see Figure 4.1). The pipeline applies the same strategy of finding a human match m_{hi} for each leftover algorithm match m_{aj} as mentioned in the last paragraph. Eventually, the pipeline collects a list of match pairs with labels. Then the pipeline calculates the precision, recall, and F_1-score as defined in Section 3.6 with the labeled match pairs as the last step in Figure 4.1.

4.2 Evaluation of Generated Statistics

This section explains how to evaluate the algorithm by comparing the statistics it produced with the same set of statistics from human graders for the domain model assessment. Beyond producing matches, the proposed algorithm goes a step further by providing a detailed set of statistics including precision, recall, and F_1-scores for each type of model element, along with final grades (refer to Section 3.6 and Section 3.7). This numerical assessment offers intuitive feedback on the domain model. Therefore, it can be reformulated as a regression problem. The algorithm should generate or predict statistics that are as close as possible to the statistics produced by human graders. To evaluate the performance of the algorithm in generating meaningful statistics, MAE is applied (see Section 2.4). Ranking students' submissions is also a common technique to reveal the relative qualities of their work. Therefore, the alignment between the ranking of domain models based on the algorithm-generated grades and the ranking based on grades from a human grader is also examined. The alignment is measured using Kendall's τ. Furthermore, numeric grades are translated into letter grades following the scheme of McGill University[1] which is commonly used in North America, and this thesis also evaluates the deviation in letter grades.

[1] https://www.mcgill.ca/study/2023-2024/university_regulations_and_resources/undergraduate/gi_grading_and_grade_point_averages

4.3 Summary

In conclusion, this chapter demonstrates how to evaluate the performance of the algorithm from two major aspects. Section 4.1 emphasize a more rigid evaluation setting of comparing algorithm-generated matches with human-grader-generated matches. Matches are evaluated in pairs to calculate the precision, recall, and F_1-score. Section 4.2 presents a loose evaluation setting by treating it as a regression problem, only focusing on the numbers it generates, while ignoring the matching part. The upcoming chapter delves into an exploration of the experiments conducted throughout this thesis.

Chapter 5

Experiments

This chapter aims to evaluate the algorithm's performance in assessing domain models and identify areas of the matching and grading task where the algorithm may encounter challenges. More concretely, this chapter aims to investigate the following research questions (RQ):

RQ 1

What is the performance of the algorithm in matching a candidate domain model to a reference model regarding classes, attributes, and relationships?

RQ 2

To what extent does the algorithm-generated grade compare with those produced by human grading?

This chapter begins by describing the experimental settings in Section 5.1. Subsequently, each research question is addressed, and the findings are reported in Sections Section 5.2 and Section 5.3. Finally, this chapter provides an in-depth analysis and discussion of the results including qualitative results in Section 5.4 and state threats to validity in Section 5.5.

5.1 Experimental Settings

5.1.1 Modeling Problem and Solution

This thesis has chosen a modeling problem within the smart home domain that originally served as an assignment in an introductory-level, undergraduate modeling course. The assignment requires students to create a domain model for a smart home automation system (SHAS) that allows various users to automatically manage smart home automation tasks. The complete textual problem description can be found in Appendix A.1. The class diagram of the reference domain model created by the course instructor, along with the textual domain model in the EBNF format can be found in Appendix A.2. There are 5 enumeration classes, 15 regular classes, 3 abstract classes, 13 enumeration literals, 13 attributes, and 32 relationships in the reference domain model.

5.1.2 Test Set

20 out of 104 student solutions to the modeling problem are randomly selected to perform an analysis of the algorithm. The author of this thesis had no relationship to the course which featured the SHAS assignment and hence had not seen the student solutions prior to the analysis. To establish the ground truth for the experiments, at first, each student's solution is converted to the customized model representation manually from the original Umple file. Then the author of this thesis manually matches the candidate model against the reference model at the granularity of the element level (see Section 4.1). The complete algorithm and supplementary materials are available[1].

5.1.3 External Libraries

The algorithm relies on some external libraries for embeddings and graph comparison. Embeddings play a fundamental role in the algorithm. There are several places where word

[1] https://github.com/ChenKua/Domain_Model_Evaluation

embeddings and sentence embeddings are needed. For the choice of word embedding, `sgram_mde` (skip-gram for MDE) from the WordE4MDE library[2] created by Hernández et al. [25] is selected. This word embedding approach can embed an individual word into a 300-dimensional vector. For the choice of sentence embedding, one of the OpenAI embedding models, `text-embedding-ada-002`, is used via the API provided by OpenAI[3], which embeds a sequence of up to 8191 tokens into a 1,536-dimensional vector. Meanwhile, calculating graph edit distance is also essential to the algorithm, which relies on the Python library NetworkX[4] [32] to find the optimal graph edit distance between two graphs.

5.2 RQ 1: Matching Performance of the Algorithm

RQ 1 aims to evaluate how the algorithm performs in matching model elements from the candidate model to elements from the reference model. This thesis is interested in how far the algorithm is from a perfect performance score for this task. This will show if there is room for improvement and whether the automated domain model assessment problem is solved with the approach. Furthermore, a more fine-grained analysis is carried out by highlighting with which modeling aspects the algorithm struggles. Particularly, this thesis compares the performances of the algorithm when matching classes, attributes, and relationships; and studies for which evaluation metrics (precision, recall, and F_1-score) the algorithm struggles.

Table 5.1 shows the individual precision, recall, and F_1-scores of matching 20 student submissions. The highest class precision, recall, and F_1-score reach 1.0, 1.0, and 1.0, while the lowest values are 0.5714, 0.8333, and 0.6857, respectively. The highest attribute precision, recall, and F_1-score achieve 0.9, 1.0, and 0.9231. The lowest ones are 0.4375, 0.5, and 0.5833. Meanwhile, the highest relationship precision, recall, and F_1-score are 1.0, 1.0, and 1.0, while the lowest ones are 0.5556, 0.3846, and 0.4545. It can be observed that the minimum values of class precision, recall, and F_1-score exceed those of attributes and relationships, respectively.

[2] https://github.com/models-lab/worde4mde
[3] https://platform.openai.com/docs/guides/embeddings/embedding-models
[4] https://networkx.org/documentation/stable/reference/algorithms/similarity.html

Table 5.1: Performance scores for matching over 20 student submissions; highest values in each column are highlighted in blue, while lowest values are highlighted in red

	Class			**Attribute**			**Relationship**		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
1	0.6818	1.0	0.8108	0.7143	0.5556	0.625	0.9286	0.7647	0.8387
2	0.9	1.0	0.9474	0.8182	0.8182	0.8182	1.0	1.0	1.0
3	0.7647	0.8667	0.8125	0.8333	0.7895	0.8108	0.8462	0.6875	0.7586
4	1.0	1.0	1.0	0.6818	0.75	0.7143	1.0	0.8462	0.9167
5	0.6667	0.8889	0.7619	0.74	0.74	0.74	0.9286	0.8125	0.8667
6	0.8095	0.9444	0.8718	0.76	0.7037	0.7308	0.875	0.6364	0.7368
7	0.6522	0.9375	0.7692	0.75	0.6667	0.7059	0.8889	0.7273	0.8
8	0.8947	1.0	0.9444	0.7143	0.75	0.7317	1.0	1.0	1.0
9	0.75	1.0	0.8571	0.9	0.8438	0.871	0.6364	0.875	0.7368
10	0.9048	1.0	0.95	0.875	0.8235	0.8485	0.8333	1.0	0.9091
11	0.5714	0.9231	0.7059	0.7308	0.5	0.5937	0.6667	0.4	0.5
12	0.7647	0.9286	0.8387	0.9	0.9474	0.9231	1.0	0.7778	0.875
13	0.6087	0.875	0.7179	0.6346	0.9706	0.7674	1.0	1.0	1.0
14	0.6667	0.8571	0.75	0.4706	0.8	0.5926	0.75	0.6667	0.7059
15	0.75	0.8333	0.7895	0.5526	0.7	0.6176	1.0	0.5	0.6667
16	0.5714	0.8571	0.6857	0.4375	0.875	0.5833	0.5556	0.3846	0.4545
17	0.7619	0.9412	0.8421	0.8913	0.8913	0.8913	0.8333	0.7692	0.8
18	0.7647	1.0	0.8667	0.6875	0.8462	0.7586	0.5833	0.875	0.7
19	0.6818	0.8824	0.7692	0.6667	0.75	0.7059	0.7857	0.6875	0.7333
20	0.6842	0.9286	0.7879	0.7895	1.0	0.8824	1.0	0.8462	0.9167

Table 5.2: Average performance scores for matching each model element

Metric		**Average ± Standard Deviation**
Class	Precision	0.7425 ± 0.1156
	Recall	0.9332 ± 0.0587
	F_1-score	0.8239 ± 0.0873
Attribute	Precision	0.7274 ± 0.1318
	Recall	0.7861 ± 0.1265
	F_1-score	0.7456 ± 0.1067
Relationship	Precision	0.8556 ± 0.1492
	Recall	0.7628 ± 0.1847
	F_1-score	0.7958 ± 0.1505

In terms of performance scores for class matching and relationship matching, the algorithm achieves a perfect performance score in certain cases. However, it is worth noting that such perfection is not attainable in attribute matching. Table 5.2 shows the average precision, recall, and F_1-scores for the matching results generated by the algorithm compared with

human-generated matches of 20 student submissions. It is evident that the algorithm excels in class matching with an F_1-score of 0.8239, surpassing those for attribute matching (0.7456) and relationship matching (0.7958). Notably, the F_1-score for attribute matching is the lowest among the three types of model elements. The precision of relationship matching is the highest among the three, while the recall of class matching highly surpasses those of the other two elements. Based on the trend of F_1-scores, it can be inferred that the algorithm excels in matching classes compared to matching relationships and it performs better in matching relationships than in matching attributes. Additionally, the recall for matching classes and attributes is notably higher than their respective precision, while the phenomenon is reversed for matching relationships. Class matching and attribute matching share a similar logic of embedding-based graph edit distance to match elements, whereas relationship matching is rule-based with dependency on class matching. Relationship matching relies on class matching, which might be why relationship matching has a higher precision than recall. This phenomenon implies a potential further optimization direction on balancing precision and recall in class matching and attribute matching. A more detailed discussion on this can be found in Section 5.4.

Answer to RQ 1. While the algorithm demonstrates impressive capability in matching model elements, there is still room for improving the performance. The algorithm achieves F_1-scores of 0.82 for class matching, 0.75 for attribute matching, and 0.80 for relationship matching. Moreover, the algorithm struggles the most with matching attributes (compared to classes and attributes).

5.3 RQ 2: Grading Performance of the Algorithm

In addition to generating element matches, the algorithm produces a set of statistics as numerical assessments for each assessed domain model. RQ 2 seeks to explore whether these statistics can serve as meaningful grades for evaluated domain models. It aims to investi-

gate the extent to which the algorithm is capable of generating statistics that reasonably approximate the ground truth values.

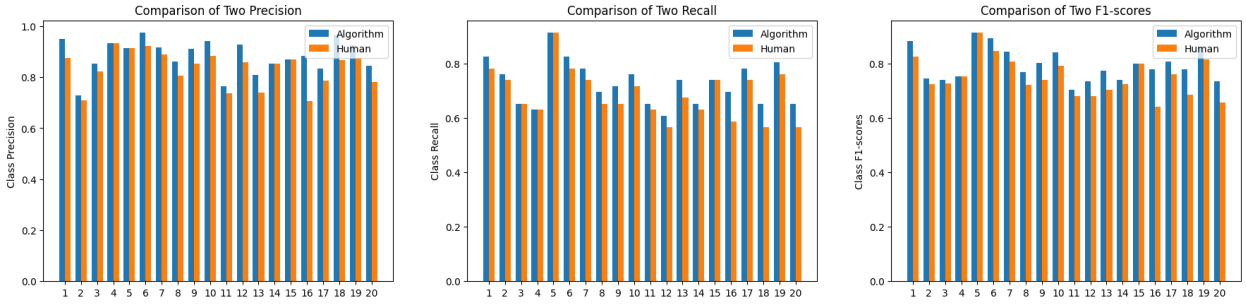
Two primary sources provide ground truth values for comparison on this research question. Initially, the author manually matches model elements and assigns corresponding matching scores. Consequently, based on these human-generated matches and matching scores, precision, recall, and F_1-scores are calculated for each assessed domain model. Leveraging these calculated F_1-scores, the final grade can be calculated using the weighted average method as previously discussed in Section 3.7. This constitutes the first type of ground truth value. Given that these values are directly from the author of this thesis, the process of comparing them against algorithm-generated data is denoted as *internal comparison*.

Meanwhile, this thesis compares the final grades generated by the algorithm with those from another existing benchmark on automated domain model assessment [8]. This serves as the second source of ground truth values. Henceforth, it shall be referred to as *external comparison*.

5.3.1 RQ 2.1: Internal Comparison

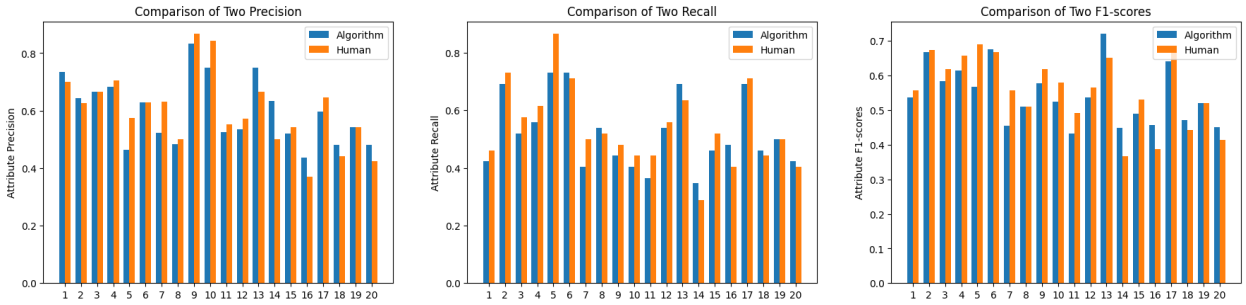
In RQ 2.1, an *internal comparison* is conducted where algorithm-generated precision, recall, F_1-scores, and grades are compared with the same set of results produced by the author of this thesis. Inferential statistics are conducted to support the results.

Figure 5.1, Figure 5.2, Figure 5.3, and Figure 5.4 demonstrates statistic results on precision, recall, F_1-scores, and grades, respectively, for each student submission. The blue bars illustrate data generated by the algorithm, while the orange bars represent data sourced from the author of this thesis. From Figure 5.1, it seems that the algorithm is always more generous in giving higher values than the ground truth values when evaluating classes. To confirm this observation, the number of domain models which receive higher algorithm-generated results compared with human results are counted as shown in Table 5.3. For class precision, recall, and F_1-scores, 20 out of 20 domain models demonstrate algorithm-generated values



(a) Class Precision **(b)** Class Recall **(c)** Class F_1-scores

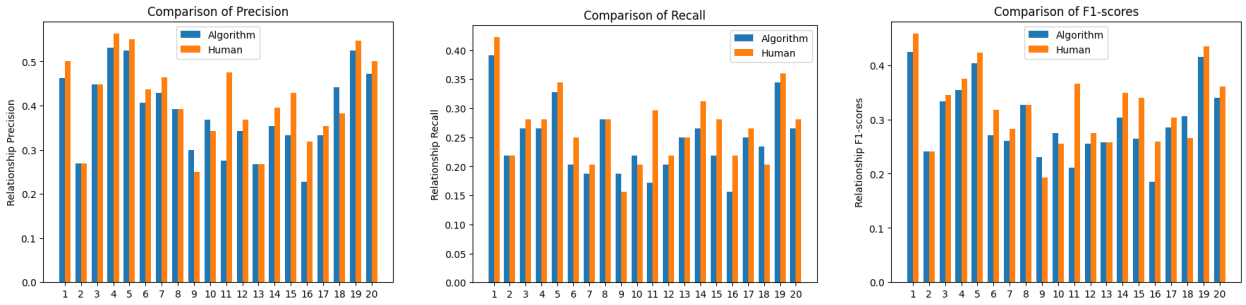
Figure 5.1: Three plots showing class precision, recall, and F_1-scores generated by the algorithm and generated by the human



(a) Attribute Precision **(b)** Attribute Recall **(c)** Attribute F_1-scores

Figure 5.2: Three plots showing attribute precision, recall, and F_1-scores generated by the algorithm and generated by the human

that are greater than or equal to their human-generated counterparts. Such a situation does not exist for attributes or relationships.



(a) Relationship Precision **(b)** Relationship Recall **(c)** Relationship F_1-scores

Figure 5.3: Three plots showing relationship precision, recall, and F_1-scores generated by the algorithm and generated by the human

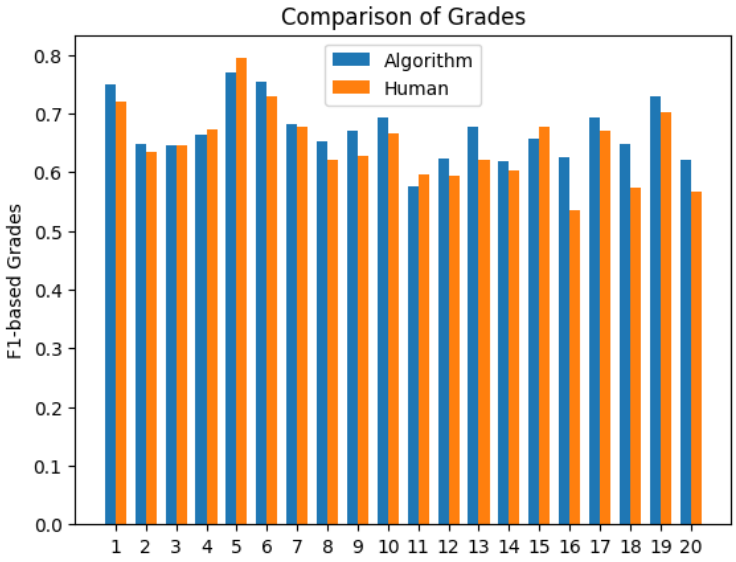


Figure 5.4: Internal comparison of grades

Table 5.3: The number of domain models which receive algorithm-generated results greater than or equal to the human-generated results

Metric		**Algorithm Results ≥ Human Results**
Class	Precision	20 / 20
	Recall	20 / 20
	F_1-score	20 / 20
Attribute	Precision	10 / 20
	Recall	8 / 20
	F_1-score	7 / 20
Relationship	Precision	7 / 20
	Recall	6 / 20
	F_1-score	6 / 20
	Grade	15 / 20

Table 5.5 demonstrates the results of various inferential statistical tests. To determine if the samples are normally distributed, the Shapiro-Wilk test [35] is employed. This test yields a p-value that guides the decision-making process: if the p-value is greater than the significance level of 0.05, the null hypothesis is not rejected, indicating that the data follows a normal distribution; conversely, if the p-value is less than 0.05, the null hypothesis is rejected, suggesting the data is not normally distributed. When the data is normally distributed, the paired T-test [36] is used to evaluate differences; otherwise, the Mann-Whitney U-test [37] is applied. A significance level of 0.05 is used for both T-tests and U-tests in this thesis. These

Table 5.4: The Mean Absolute Error (MAE) between algorithm-generated data and author-generated data; MAE values closer to 0 signify better performance, while a correlation (Pearson correlation) approaching 1 indicates a stronger alignment between the datasets

Metric		**MAE**	**Correlation**
Class	Precision	0.04923	0.8146
	Recall	0.04130	0.9418
	F_1-score	0.04507	0.8702
Attribute	Precision	0.04723	0.8702
	Recall	0.04712	0.9193
	F_1-score	0.04427	0.8439
Relationship	Precision	0.04110	0.8272
	Recall	0.02891	0.8306
	F_1-score	0.03397	0.8006
	Grade	0.03096	0.8714

tests output a p-value, which is then used to determine whether to reject the null hypothesis. If the observed p-value is less than 0.05, it provides strong evidence to reject the null hypothesis. The null hypothesis typically suggests that there is no significant difference between groups, or that any observed effect is likely due to random chance. Therefore, rejecting the null hypothesis indicates that there is a significant difference between the data. Conversely, if the p-value exceeds 0.05, there is insufficient evidence to reject the null hypothesis, suggesting that any difference between the two groups may not be statistically significant. In this experiment, the algorithm is expected to grade domain models in a manner consistent with human graders. Ideally, the metrics or grades it generates should closely align with those given by human graders, without showing any statistically significant differences. Our analysis reveals that out of ten evaluated metrics, only three show no statistically significant difference between the algorithm's results and human evaluations. Specifically, there is no significant difference in the three metrics related to attribute matching, indicating that the algorithm performs comparably to human graders in this area. However, significant differences are observed in metrics related to class matching, relationship matching, and the final grade, highlighting discrepancies between the algorithmic assessments and those made by human graders. These findings suggest that while the algorithm shows potential, it currently cannot replace human graders and requires further improvement.

Table 5.5: Inferential statistics for algorithm and human grading results; Normality refers to if data are normally distributed; SD refers to if there is a statistically significant difference; the p-values in the Algorithm and Human columns are derived from the Shapiro-Wilk test for normality for each data group; the p-values in the T-test / U-test column are derived from either a T-test or a Mann-Whitney U test (with the U-test specifically applied to the Attribute-Recall row)

		Algorithm		**Human**		**T-test / U-test**	
		P-Value	Normality	P-Value	Normality	P-Value	SD
Class	Precision	0.2621	✓	0.1066	✓	0.00004	✓
	Recall	0.3235	✓	0.1710	✓	0.000009	✓
	F_1-score	0.2022	✓	0.6986	✓	0.00001	✓
Attribute	Precision	0.2326	✓	0.7323	✓	0.7569	×
	Recall	0.0398	×	0.4731	✓	0.5692	×
	F_1-score	0.2284	✓	0.1876	✓	0.2383	×
Relation-ship	Precision	0.4945	✓	0.5929	✓	0.03809	✓
	Recall	0.2661	✓	0.4167	✓	0.01577	✓
	F_1-score	0.4483	✓	0.7874	✓	0.02155	✓
	Grade	0.4376	✓	0.9531	✓	0.002873	✓

Table 5.4 shows the mean absolute error (MAE) (refer to Section 2.4) between precision, recall, and F_1-scores generated from the algorithm compared with the same metrics from a human grader. The MAE consistently remains below 0.05, which is 5% in terms of percentage. In many universities, a 5% grading scale range is commonly used for undergraduate course assessments. For example, an A- is typically assigned to numerical grades ranging from 80% to 85%, while a B+ corresponds to grades between 75% and 80%. The algorithm combines three F_1-scores through weighted averaging to derive the final grade. Employing identical weights as those utilized in the algorithm, the weighted average of F_1-scores from human graders is computed, thereby establishing them as the ground-truth F_1-based grades. Consequently, the MAE between the algorithm-generated grades and the ground truth F_1-based grades are calculated, as shown in the final row of Table 5.4. This computed MAE stands at 0.03096, indicating a small deviation. Furthermore, the correlation between metrics derived from the algorithm and those from human graders is computed and exhibited in the last column of Table 5.4. All correlation coefficients surpass 0.8, indicating a robust correlation between metrics generated by the algorithm and those by human graders.

Evaluating students' submissions through ranking is a widely adopted practice in education. Consequently, the rankings of the 20 student submissions are calculated based on grades generated by the algorithm and those from human graders. To measure the alignment in ranking, the Kendall τ coefficient is employed (refer to Section 2.4). The resulting Kendall τ coefficient between the two rankings stands at 0.6526 with a p-value near zero, signifying a robust alignment between the two rankings.

Answer to RQ 2.1. The algorithm excels in accurately evaluating domain models, demonstrating precision, recall, F_1-scores, and grades that closely align with the ground truth values. Across all metrics, the Mean Absolute Error (MAE) remains consistently below 0.05, accompanied by strong correlations exceeding 0.8. However, only the metrics related to attributes but not the metrics related to classes, relationships, and the overall grade show no statistically significant difference between the algorithm's results and human evaluations, indicating that there is room for improvement.

5.3.2 RQ 2.2: External Comparison

RQ 2.2 aims to evaluate how well the algorithm produces a grade for an assessed domain model that approximates an existing external benchmark from Singh et al. [8].

Table 5.6: Grading scheme

Letter Grade	Integer Value
A	10
A-	9
B+	8
B	7
B-	6
C+	5
C	4
D	3
F	2

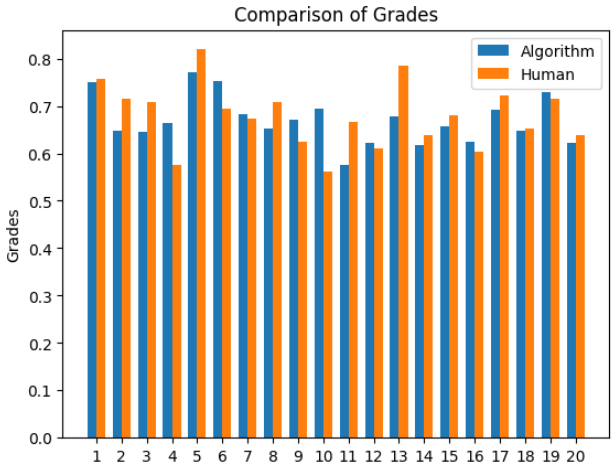


Figure 5.5: External comparison of numerical grades

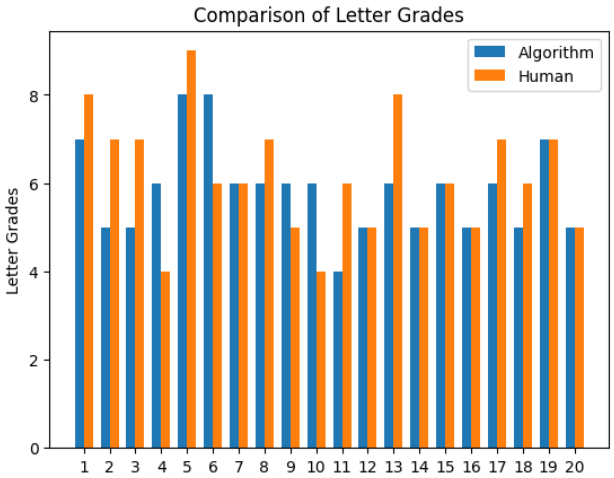


Figure 5.6: External comparison of letter grades

The comparison is conducted on two levels: the first level involves comparing numerical grades ranging from 0 to 1 in decimals, while the second level entails comparing letter grades. The algorithm-generated grades are compared with the same set of domain model grading results in the benchmark as shown in Figure 5.5. Inferential statistics are also conducted to compare algorithm results with the external benchmark. The Shapiro-Wilk tests confirm that both groups of data are normally distributed. Consequently, a paired T-test is performed, yielding a p-value of 0.5613. This result indicates that there is no significant difference between the two groups of data, highlighting the impressive performance of the algorithm in generating results that closely align with the external benchmark. The numerical grades

for the same set of student submissions are obtained and the MAE between their grades and ours are calculated. The largest difference in numerical grades (absolute error) among the domain model assessments is 0.1317, while the smallest difference is 0.004. The resulting MAE of *0.0456* is deemed reasonably small and falls within the spectrum of a typical letter grade range.

Additionally, both sets of numerical grades are converted into letter grades using the grading scheme adopted by McGill University[5] for undergraduate courses. Subsequently, integers are assigned to the letter grades according to the scheme in Table 5.6, where adjacent letter grades differ by 1, allowing the calculation of the MAE between the two sets of letter grades. The letter grades for each domain model in two assessments are presented in Figure 5.6. The largest difference in letter grades is 2, while the smallest difference is 0. There are 7 out of 20 domain models whose letter grade difference is exactly 0. There are 6 out of 20 domain models with a letter grade difference of 1, and 7 out of 20 domain models with a letter grade difference of 2. The MAE is exactly 1. This indicates a reasonably close resemblance between the grades generated by the algorithm and those given by human graders, although there is still room for improvement.

Answer to RQ 2.2. The algorithm showcases its proficiency in producing meaningful precision, recall, F_1-scores, and grades for domain model assessments, which closely approximate those of the external benchmark. The Mean Absolute Error (MAE) between the data and the benchmark stands at 0.0456 for numerical grades and 1 for letter grades. The t-test confirms the absence of a statistically significant difference between the grades produced by the algorithm and those from the benchmark.

[5] https://www.mcgill.ca/study/2023-2024/university_regulations_and_resources/undergraduate/gi_grading_and_grade_point_averages

5.4 Discussion

This section discusses several aspects of the experiments and possible implications for each research question.

5.4.1 RQ 1

The experiments show that the algorithm is still not able to match the model elements completely the same as a human grader. However, the results obtained are promising and there is still room for improvement. The results in RQ 1 reveal that the algorithm struggles to match attributes correctly. Meanwhile, class matching and attribute matching, both suffer from lower precision and higher recall. This suggests a higher percentage of false positives compared to false negatives in the matches. Based on the discussion of FP (refer to Table 4.2), the situation for FP can be summarized. Ideally, a model element `a` is matched with `b` as a ground truth match `(a, b)`. The element `a` cannot be `None` but `b` can be `None`. The reason for a false positive is that the algorithm mistakenly matches `a` with some elements other than `b`. More specifically, there can be two cases depending on the element `b`: (1) `b` = `None`, and `a` is matched with `c` where `c` $\neq$ `None`, or (2) `b` $\neq$ `None`, and `a` is matched with `c` where `c` $\neq$ `b` and `c` $\neq$ `None`. Therefore, one future direction can be aiming to reduce the false positives by better identifying `None` matching by defining more rule-based or embedding-based mechanisms to identify `None` matching, i.e., `(a, None)`. At the same time, given the algorithm is a multi-stage approach, its behavior in an interactive setting can also be studied. In this case, a human grader can continuously provide feedback or tweak the generated matches at each stage to continuously improve the overall performance.

5.4.2 RQ 2

By addressing the second research question, this thesis aims to provide insight into the extent to which the algorithm produces statistics that reasonably approximate the ground truth. This thesis performs an internal comparison and an external comparison. For internal com-

parison, the algorithm tends to be overly generous in its evaluations of classes. Specifically, all domain models assessed by the algorithm receive higher class precision, recall, and F_1-scores than their human-generated counterparts. This may imply the algorithm needs higher thresholds or other mechanisms to reduce false positives. Nevertheless, through meticulous evaluation and comparison, it can be observed that the algorithm's outputs exhibit a high degree of correlation with the ground truth values. The Mean Absolute Error (MAE) for the ten metrics related to class matching, attribute matching, relationship matching, and final grading is less than 0.05, indicating a small error relative to human evaluations. Additional statistical tests were conducted to determine whether there is a significant difference between the algorithm's results and human evaluations. However, only three out of the ten metrics pass the statistical tests, failing to reject the null hypothesis and showing no statistically significant difference in the algorithm's grading and human grading. The three metrics showing no significant difference are related to attribute matching, which had the lowest performance in RQ 1. This contrast may be because RQ 1 focuses on evaluating individual matches, while RQ 2 assesses the overall score. Therefore, errors in individual matches may offset each other, resulting in a more balanced overall score. Conversely, the remaining seven metrics fail to pass the statistical tests and exhibit statistically significant differences in the grading. This indicates that, although the algorithm demonstrates impressive capability in grading domain models, it cannot currently replace human graders and requires further enhancement. For the external comparison, the MAE between the numerical grades from the algorithm and the external benchmark is less than 0.05. The results from the additional T-test also confirm that there is no statistically significant difference between the two groups of data. When the grades are converted to letter grades, the MAE becomes 1, indicating a difference between the grades.

Overall, the algorithm demonstrates its effectiveness by generating meaningful precision, recall, F_1-scores, and grades that closely align with the ground truth values internally and externally, indicating its capability to accurately assess domain models. This alignment between the algorithm's outputs and the ground truth values highlights its effectiveness

and potential in providing insightful assessments of domain models. Nevertheless, seven statistical tests indicate a significant difference between the algorithm's grading and human grading. Besides, the algorithm is unable to predict letter grades to the same extent as human grading. This implies that the algorithm cannot fully replace human grading in real-world scenarios and underscores the need for further improvement to reach that level of expertise.

5.5 Threats to Validity

5.5.1 Internal Validity

The matching dataset is constructed by the author, which may introduce bias. This thesis mitigates this bias by comparing the algorithm outputs with a human-grading benchmark created by another set of researchers [8]. This benchmark includes the assignments used in this thesis. Meanwhile, the selection of scores for model elements and weights for F_1-scores can influence the final grades. The scores follow our previous work and the weights are chosen based on those commonly used to evaluate university-level assignments. There are typically multiple ways to represent a domain model, which may influence the performance of the algorithm. To be consistent with our previous work, this thesis follows the same text-based domain model representation that is based on natural language and uses it across all experiments.

5.5.2 External Validity

There are benchmarks [8] for grading in the automated domain model assessment research. However, to the best of my knowledge, there are no benchmark data sets available for model element matching. Therefore, the author of this thesis curates such a data set manually. This thesis only includes a limited number of data points in the evaluation of the algorithm, which may increase the risk of obtaining false positives in the statistical tests. Furthermore, models

are collected from an undergraduate modeling course representing modeling in education scenarios. The algorithm may perform differently if used in a different scenario. Many third-party libraries are used in the algorithm, specifically on embedding approaches. There could be risks regarding unexpected changes or updates of OpenAI embedding approaches. The OpenAI embedding result of model elements used in the experiments has been saved in a public GitHub repository[6] to ensure reproducibility and enable future researchers to validate and build upon the findings. Additionally, the letter grade range is based on the system used by McGill University, which may limit generalizability to other letter grade systems.

5.5.3 Construct Validity

The selection of metrics is important to reflect the performance of the algorithm. For instance, using precision and recall for a regression problem cannot correctly evaluate the performance of the model. There are various metrics employed across different sections of this thesis. To evaluate the matching performance of the algorithm, this thesis adapts metrics widely used for evaluating domain models [2, 8, 38] including precision, recall, and F_1-score. To evaluate the grading performance of the algorithm, this work adapts MAE [39, 40] and correlation [41, 42] which are also commonly used in the evaluation methods for regression problems.

5.6 Summary

In summary, this chapter answers the proposed research questions and evaluates the performance of the algorithm. It begins by describing the experimental settings in Section 5.1, where evaluation datasets and external libraries are introduced. Then, Section 5.2 investigates RQ 1, i.e., the capability of the algorithm to match model elements. The findings reveal that the proposed algorithm is impressive in matching model elements with F_1-scores of 0.82 for class matching, 0.75 for attribute matching, and 0.80 for relationship matching. Yet, it

[6] https://github.com/ChenKua/Domain_Model_Evaluation

is still far from a perfect expert on matching model elements. In Section 5.3, RQ 2 proposes two sub-questions, addressing internal and external comparisons, respectively. In both scenarios, the algorithm showcases its proficiency in assessing domain models, yielding metrics that closely correspond to the ground truth values. The mean absolute error (MAE) remains consistently below 0.05 across all metrics, affirming the algorithm's accuracy. During the internal comparison, the correlation between the algorithm-generated metrics and those from a human grader consistently exceeded 0.8. The Kendall τ coefficient reaches 0.65, indicating a strong alignment between the rankings of student submissions derived from algorithm-generated grades and those from human grading. The algorithm performs well in producing metrics for attribute matching, showing no statistically significant difference compared to human grading. However, it struggles with class matching and relationship matching, where statistically significant differences are observed between the algorithm's results and human evaluations. The algorithm generates final grades with a statistically significant difference compared to the grading provided by the author of this thesis but shows no statistically significant difference when compared to the external benchmark. The MEA of 1 in letter grades suggests a remarkable performance of the algorithm in assigning letter grades reasonably close to human grading. However, it still needs further improvements to replace human grading.

Section 5.4 discusses the implications of each research question, while Section 5.5 illustrates threats to validity. The next chapter discusses the related work of this thesis.

Chapter 6

Related Work

This chapter delves into the existing literature concerning the automated evaluation of domain models, as detailed in Section 6.1. Furthermore, we explore the intersection of Natural Language Processing (NLP) research with Model-Driven Engineering (MDE), as highlighted in Section 6.2 where the first two subsections talk about automated domain modeling and goal modeling, and the last subsection focuses on Large Language Models (LLMs). The next section Section 6.3 demonstrates related research endeavors undertaken throughout my master's academic journey.

6.1 Domain Model Evaluation

Evaluating generated domain models can be challenging due to the variety of model elements and intricate design patterns involved. Many approaches have been investigated to assess the candidate solution in comparison to the reference solution. A table summarizing the papers with techniques behind the scenes is shown in Table 6.1

Rule-based Approaches. Yang and Sahraoui [10] combine learning-based and rule-based approaches for the extraction of UML class diagrams from natural language software specifications. Yet, the evaluation of extracted UML class diagrams is still rule-based. Specifically, they implemented an NLP pipeline that uses a trained classifier to tag each sentence

Table 6.1: Summary of automated domain model assessment

Paper	Heuristics / Rules	Graph Matching	Machine Learning	
			Training	Embedding
Yang and Sahraoui [10]	✓			
Bien et al. [11]	✓			
Jayal and Shepperd [12]	✓			
Hasker [13]	✓			
Tselonis et al. [14]	✓	✓		
Ludovic et al. [15]	✓	✓		
Boubekeur et al. [9]	✓		✓	
Singh et al. [8]	✓			
Our proposed approach	✓	✓		✓

from the specification as describing either a class or a relationship, mapped each sentence into a UML fragment, and assembled the fragments into a complete UML diagram using a composition algorithm. In their evaluation, they evaluate the result with reference solutions using exact matching, relaxed matching, and general matching over classes and relationships. However, in the evaluation, they do not deal with the synonyms issue in the generated solution, resulting in low performance of their approach. Bien et al. [11] present an approach for automated grading of UML class diagrams, which uses a grading algorithm with syntactic, semantic, and structural matching between two class diagrams. A metamodel was introduced to store mappings and grades for mapping each model element, e.g., classes, attributes, and associations so that it is possible to update the grading scheme later on. Jayal and Shepperd [12] explore the problem of class name matching with various test transformations including: trimming, stemming, lowercase, replacing punctuation characters on class names etc. Hasker [13] proposes a rule-based approach named UMLGrader to grade UML diagrams. Some rules include: *Class names only match if they match the whole name*, *Spaces and underscores are stripped before matching names*, and *capitalization is ignored*, etc. Singh et al. [8] introduce a Mistake Detection System (MDS) designed to identify errors and offer feedback to students by comparing their submissions with a solution. This system is able to detect a wide range of potential pre-defined mistakes (e.g., plural class name violation, composition violation) present in a submission. This work also focuses on matching attributes

and relationships. The core mechanism of their system is identifying the matching of class names between a student solution and an instructor solution through rule-based approaches, such as exact match, attribute overlap, Levenshtein distance (i.e., the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into the other [43]).

Graph-based Approaches. Tselonis et al. [14] propose the idea of treating various types of diagrams including UML class diagrams as graphs and then using graph matching algorithms to measure the similarity between such translated graphs. Diagrams are nothing but boxes linked by connectors, which can be easily converted to graphs. More importantly, graphs can be matched against each other for similarity, and the results used in various ways. One of the matching algorithms investigated by the author is graph isomorphism. Two graphs are isomorphic if we can change the vertex labels on one to make its set of edges identical to the other [29]. In this context, a correct answer would most likely be isomorphic to, or at least contain an isomorphic subgraph of, a model answer [14]. Similarly, Ludovic et al. [15] also apply a graph-matching approach for assessing class diagrams. Their proposed algorithm is based on graph-matching techniques, using characteristic structural patterns depicted in diagrams. Similarity functions compare these structures, generating similarity scores for each pair. The algorithm categorizes matches into univalent and multivalent based on a taxonomy of differences.

Machine Learning Approach. Boubekeur et al. [9] propose an approach based on a simple heuristic and machine learning that helps categorize simple domain model submissions from students according to their quality. The system determines if submissions are above a quality threshold to assign them a letter grade. The predicted letter grades are comparable with the letter grades of the human graders. Still, their approach lacks explainability and does not give a specific score for each submission or contain any detail for each component within the system.

In comparison to existing methodologies, this thesis introduces an algorithm utilizing word embedding and sentence embedding for automated domain model assessment by assigning scores for each model element and calculating the precision, recall, and F_1-scores for classes, attributes, and relations individually. To the best of my knowledge, this is the first kind of approach which applies a combination of embeddings for automatic assessment of domain models.

6.2 NLP for MDE

Extensive downstream applications from the field of NLP have been directly applied to model-driven engineering. A table summarizing different NLP research applied in MDE is shown in Table 6.2. Numerous MDE practices involve the extraction of information from natural language documents to create abstract models, and here, the contributions of NLP research are substantial. Named Entity Recognition (Named Entity Recognition (NER)), a key NLP technique aiming to automatically identify and categorize text entities into predefined labels [44], plays a crucial role in extracting potential classes and relationships from documents. For instance, Gala [45] effectively employs NER in the generation of class diagrams from user requirements. Another relevant NLP research area is part-of-speech tagging, a process involving the automated assignment of part-of-speech tags (such as verbs, adjectives, adverbs, nouns, etc.) to words in a sentence [46]. This technique also contributes significantly to the information extraction processes within the context of MDE. Recently, with rapid advances in auto-regressive language models, LLMs have shown powerful generalizability to tasks beyond NLP [4]. LLMs can perform different tasks without supervised training on the specific task using carefully designed input (called prompt). Using different *prompt engineering* [5] techniques, LLMs can achieve impressive performance on different tasks by only using a few labeled examples in the prompt. Such advances raise the natural question of whether these LLMs can be used to *fully automate* domain modeling. Therefore, our previous research evaluated to what extent *domain modeling* and *goal modeling* can be

Table 6.2: Summary of NLP research used for MDE; labels in the second column indicate: Part-of-speech Tagging (POS), Named Entity Recognition (NER), Embeddings (E), Large Language Model (LLM); *Generation* indicates complete model generation and *Assistant* indicates modeling assistant

Paper	NLP Research	Method	Domain Modeling		Goal Modeling	
			Generation	Assistant	Generation	Assistant
[38]	POS	Rule-based	✓			
[47]	POS	Rule-based	✓			
[48]	POS	Rule-based	✓			
[49]	POS	Rule-based				✓
[50]	POS, LLM	Rule-based				✓
[51]	POS	Rule-based	✓			
[52]	NER	Rule-based	✓			
[45]	NER	Statistical	✓			
[53]	E	Statistical		✓		
[10]	E	Statistical	✓			
[54]	LLM	Statistical		✓		
[2]	LLM	Statistical	✓			
[6]	LLM	Statistical		✓		
[55]	LLM	Statistical		✓		
[56]	LLM	Statistical		✓		
[57]	LLM	Statistical				✓
[7]	LLM	Statistical			✓	

fully automated using an LLM *without* supervised training [2, 7]. The work on goal modeling [7] explores GPT-4's current knowledge and mastery of a specific modeling language, the Goal-oriented Requirement Language (GRL). We also conduct case studies using GPT-4 to create goal models. Our results suggest that GPT-4 preserves considerable knowledge on goal modeling, and although many elements generated by GPT-4 are generic, reflecting what is already in the prompt, or even incorrect, there is value in getting exposed to the generated concepts. The work on domain modeling [2] evaluates how the expressiveness of LLMs and different prompt engineering techniques can affect the quality of generated domain models. Furthermore, we identified the advantages and limitations of current LLMs for fully automated domain modeling. More importantly, our experiments in this research suffered from the lack of a proper automated domain model assessment tool, which directly motivated this thesis.

6.2.1 Automated Domain Modeling

Many existing works on automated domain modeling utilize statistical methods or rule-based methods to directly derive complete domain modeling solutions like UML class diagrams [10,38,47,51] or provide modeling assistance and suggestions [53,54] from textual descriptions in natural language. Table 6.2 summarize different methods for automated domain modeling.

Statistical methods emphasize using NLP techniques to extract domain models. Burgueño et al. [53] design a framework to suggest new model elements for a given partially completed model, by using word embeddings to capture the lexical and semantic information from textual documents. Several other existing approaches also combine NLP and machine learning techniques to automate the model creation process [10,38].

Rule-based methods include using hand-written grammatical templates and heuristics. An example of a rule-based method presents an algorithm with 23 heuristics which predefine a particular syntactic structure to follow and automatically identify model elements from user stories [47]. Another example of such rule-based methods is proposed by Herchi et al. [48]. The authors begin by using an NLP toolkit including a sentence splitter, tokenizer, and syntactic parser to decompose the input text and then use linguistic rules (e.g., *All nouns are converted to entity types*) to extract UML concepts.

These studies focus on generating domain models with new techniques and primarily rely on manual evaluation to assess their proposed approaches. This reliance motivates this thesis work to develop an automated domain model assessment system to replace the tedious process of manual evaluation.

6.2.2 Automated Goal Modeling

Güneş et al. [49] construct a goal model from a set of user stories by using NLP techniques to extract role names, actions, and benefits information from user stories, and then combine this information in different ways to build goal models. Zhou et al. [57] present an interactive and iterative modeling approach that merges human decision-making with deep learning, specif-

ically BERT. This approach reduces goal modeling costs while maintaining model quality. Through interviews, the authors identified practical needs of goal modelers for automating modeling using the iStar goal modeling notation [58]. Based on these findings, the proposed hybrid approach combines deep learning-based entity and relational extraction with logical reasoning using dependency and statistical rules. Wu et al. [50] propose an approach to generate iStar goal models [58] from user stories. The first step is node identification, where NLP techniques are used to extract 'who', 'what', and 'why' components from user stories. Node merging is then performed using BERT to calculate node embeddings. Pairs of nodes with a cosine similarity score above a certain threshold are merged. Finally, various kinds of relationships between nodes are identified based on predefined rules.

6.2.3 Large Language Models

Language modeling is a traditional task in natural language processing that aims to estimate the conditional probability of a sequence of tokens. Depending on the pre-training task, they can be roughly classified into the following categories [59]: *autoregressive language models*, *masked language models*, and *encoder-decoder language models*. Autoregressive language models' objective is to learn to predict the next word given the previous words of a sentence (e.g., GPT-3 [5], GPT3.5 [60][1], Codex [61]). Masked language models are trained with the objective to predict a masked word given the rest of the sequence as context (e.g., BERT [26], RoBERTa [62]). Finally, encoder-decoder language models can be trained on a sequence reconstruction or translation objective (e.g., T5 [63], BART [64]). Recently, large language models (LLMs) have gained significant attention for this task. LLMs use deep neural networks, typically with transformer architecture [65] and pre-trained in an autoregressive manner, to estimate this probability distribution.

Given a sequence of tokens $s = \{s_1, s_2, ..., s_{k-1}\}$, LLMs estimate the conditional probability of the next token $P(s_k|s_1, ..., s_{k-1})$. These models are typically used for text generation

[1] https://openai.com/blog/chatgpt

through auto-regression. Specifically, in each time step, the LLM predicts a new token that is added to the input.

Traditional NLP systems were initially designed to be task-specific, that is, trained from scratch using specific data and training objectives. However, this paradigm has changed, and currently, the state-of-the-art results are achieved by pre-trained LLMs. Such huge deep learning models (i.e., models with an extensive number of parameters) are pre-trained with a large corpus (pre-training phase). Then, those models are usually fine-tuned using a particular data set and task. After the fine-tuning phase, the generalization capabilities of those models are impressive [59]. As the scale of LLMs (i.e., the number of parameters) increases, some models can be fine-tuned to perform other specific tasks beyond text generation [66].

Fine-tuning and Prompting

Fine-tuning of an LLM involves updating some of the parameters to fit a given task and a given data set [59]. This fine-tuning approach enhances the generalization ability of the trained models [59] but has two major drawbacks: (1) the training process can be computationally expensive, and (2) some tasks may not have sufficient training data for fine-tuning. Recently, LLMs have been used as few-shot learners [5], where the training examples are used as input rather than parameter updates. As an alternative to fine-tuning, prompt-based learning methods [67] involve constructing examples used as input (i.e., prompt) to LLMs. Research has shown that, when trained on a sufficiently large corpus, an LLM has the capacity to preserve a substantial amount of knowledge implicitly within its parameters [68]. The resulting LLM can be queried for different kinds of knowledge and can answer questions in a domain without further fine-tuning or training but through prompt engineering [69, 70].

Given an input and a set of examples, a typical prompt-based learning method consists of three steps [67]: (1) transform the input and the examples into task context using a template that contains unfilled slots (a process known as *prompt engineering*), (2) fill the slots with an LLM to generate an output text, and (3) extract the final output from the output text using a post-processor. In this approach, the parameters of an LLM are fixed. Depending

on the prompt engineering strategy, either a small number of labeled examples are needed, or no labeled examples are required.

LLMs for MDE

With the advancement of LLMs, various language models have been incorporated within model-driven engineering (MDE), yielding significant improvements in various aspects of the development process.

Weyssow et al. [54] propose a learning-based approach to recommend relevant domain concepts to a modeler during a meta-modeling activity by training a deep-learning model. Specifically, the authors take a RoBERTa model with weights randomly initialized and train it on thousands of independent meta-models. Their experiments show that the trained model can learn meaningful domain concepts and relationships and recommend relevant classes, attributes, and associations. Chaaben et al. [6] propose an approach for model completion by generating related elements using GPT-3. They formulate examples using classes and their relationships with other classes and then create prompts with few-shot learning. They also rank the generated classes by frequency in multiple runs. Others discuss the potential of ChatGPT in software engineering [55,56]. For example, Cámara et al. [55] present the use of ChatGPT to build UML class diagrams enriched with OCL constraints in an interactive mode.

The evaluation of all of these approaches requires manual matching in the absence of reliably automated matching. LLMs often fail to accurately match model elements and may omit certain elements during the text generation process. This limitation motivated the adoption of a more structured, matching-based approach. Consequently, the algorithm proposed in this thesis integrates word embeddings, sentence embeddings, and graph-matching mechanisms to match model elements, ultimately moving closer to automated domain model assessment.

6.3 Use Cases of Knowledge Representation

Knowledge representation is the study of how to put knowledge into a form that a computer can reason with [71]. In the era of Good Old-Fashioned AI, rule-based approaches or symbolic-based approaches like first-order logic were popular in the field of knowledge representation. However, with the increasing popularity of machine learning, the trend has been shifted to the embedding-based approaches. Many downstream applications of this are closely related to software engineering and domain modeling because they can help software engineers understand the system and develop software systems with written documentation more efficiently. In this section, we present some use cases of knowledge representation which we have dived into during our previous research. These use cases in the fields involve certain types of embedding-based techniques, which have further inspired the proposed approach in this thesis of applying word embeddings and sentence embeddings for automated domain model assessment.

6.3.1 Knowledge Graph for Explainable Information Retrieval

Text-based Information Retrieval (IR) is the task of accessing the most relevant documents, given a query in natural language as input. Modern IR systems usually exploit text embeddings, which map text into a vector such that the distance between the vectors represents the similarity of the text [18]. Thanks to recent advancements in machine learning, embedding-based methods have been adopted in many modern information retrieval systems. One of the most famous examples of IR systems is the search engine [72], which most software engineers rely heavily on to solve technical issues encountered during the software development process. While showing promising retrieval performance, these embedding-based approaches typically fail to explain why a particular document is retrieved as a query result to address explainable information retrieval, because embedding is difficult for humans to understand. In real-world IR use cases, a user can make better decisions if an additional explanation is provided for the retrieved results. For example, when a user queries what contributes

to heart disease, an explanation stating the reason helps the user decide the relevance of a result.

Extensive research has been conducted to investigate the domain of explainable information retrieval. Explainability in information retrieval can be categorized by the form of the outcome: highlighting-based, feature-importance-based, rules-based, and mixed approaches [73].

One famous example for *highlighting-based* explanation is Google Search, where content snippets and keywords are shown along with the search result. Similarly, Chios and Verberne [74] proposed highlighting most important snippet for deep IR models. *Feature importance based* explanation methods may calculate scores from queries and documents [74–76]. Some approaches operate on term importance from the query, while others focus on re-ranking the documents with explainable features [76]. *Rule-based explanation* aims to answer specific properties about the internal mechanism of the IR model [77] by providing a simplified model to explain decisions. Finally, *mixed* explanation combines all methods to provide comprehensive explainability [74, 76].

While explainable IR is mostly post-hoc, there also exist approaches using explainable embedding to rank documents [78, 79] such that the distance in the vector space explains the relatedness of the query and document.

A Knowledge Graph (KG) is a directed labeled graph where nodes represent entities, and edges capture relationships between nodes. Therefore, KGs record structured information about entities and inherently explainable relationships. KGs provide explicit semantic relations between entities, which can serve as an explanation, or help clarify the intent of the query. Normally, such semantic relations also appear in the text. For example, the sentence "obesity contributes to heart disease" reflects two entities *obesity* and *heart disease* related by the *contribute* relation.

Considerable research efforts have been devoted to the utilization of knowledge graphs for information retrieval. Meanwhile, KGs have been shown to be effective in improving the performance of information retrieval systems [80] and generating explanations for other ma-

chine learning algorithms [81]. However, the use of KG for explainable information retrieval is still restricted to domain-specific use cases [82,83].

Entities from a knowledge graph can be used within an IR system in order to help understand a user's intent, queries, and documents beyond what can be achieved through word tokens on their own [80]. Reinanda et al. [80] summarised approaches on how to leverage entity-oriented information in KG: *expansion-based*, *latent factor modeling*, *language modeling*, and *deep learning approaches*. At the same time, KG is also being used in some domain-specific applications for explanation of IR systems.

Expansion-based approaches enrich entity-oriented information in the retrieval process by expanding queries and/or documents [84], for example, expanding the query with synonyms.

Latent factor modeling attempts to find concepts in queries and documents. Xiong et al. [85] presented a new technique for improving ranking using external data and knowledge bases. This technique treats the external objects as a latent layer between query and documents to learn judging document relevance.

Language modeling approaches consider semantic information when building language models of queries and documents. For example, Ensan et al. [86] proposed a document retriever which uses semantic linking systems for forming a graph representation of documents and queries, where nodes represent concepts from documents and edges represent semantic relatedness between concepts.

Deep learning approaches use knowledge graph embeddings in neural ranking systems. Examples are embedding queries and documents in the entity space [87] and constructing an interaction matrix between queries and entity representations [88,89].

KG are used for Explainable IR. Hasan et al. [83] proposed a framework for generating explanations in domain-specific IR applications by incorporating domain knowledge graphs. Yang [82] applies KG to explain the IR model by building knowledge-aware paths with the help of attention scores. Similarly, Xian et al. [90] design an explainable recommendation system which contains explicit reasoning with KG for decision-making to make recommendations explainable.

A well-known challenge in the field is linking entities and relationships, i.e., identifying and connecting the matching parts of a KG and a piece of text [91, 92]. While existing entity linking tools can be used as a starting point, improving the explainability of IR by using KGs also necessitates (1) a general integrated architecture and (2) a detailed recipe on how to use KG information to improve specific types of explanations.

Therefore, our previous work proposed a novel general architecture for incorporating knowledge graphs for explainable information retrieval in various steps of the retrieval process [18]. We instantiate the general architecture in two parts for explainable IR by integrating an open knowledge graph Wikidata and various entity linking approaches. The first part applies KG to identify the most important sentence of a passage, which explains how the passage is related to the query. Specifically, we use extra information about the linked entity from the KG to perform query expansion in several different ways to better identify the most important sentence. The second part incorporates the idea of re-ranking documents along with explainable features to better explain the ranking of a document. We propose a new explainable feature used for re-ranking candidate documents based on the KG with entity matching. Moreover, we conduct initial experimental evaluations on the two specific explanation approaches on two datasets: WIKIQA [93] and Robust04 [94]. Assuming an ideal entity matcher, our technique improves the performance of identifying the most important sentence by 6.96% and the base retrieval performance by 0.45%.

Explainability is also a key factor in domain model assessment. In real-world assignment grading, instructors not only assign a grade to the domain model but also provide explanations or highlight the reasons for score reductions. In the early phase of this work, LLMs are used to grade domain models against a reference model. Although LLMs can generate free-form text feedback, this feedback is often too generic. This inspired the author to explore how explainability can be derived at the model element level. Therefore, this thesis work explains the grading by the matching of model elements. Additionally, the investigation into various sentence embedding models further inspires the use of text embeddings for matching model elements in this thesis work.

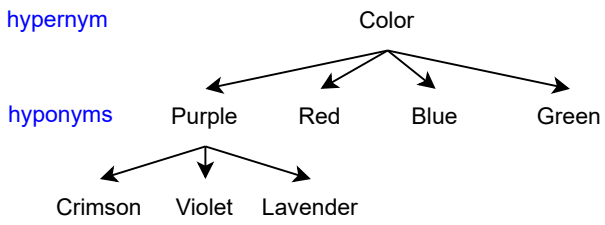


Figure 6.1: An example of the hierarchical relationship between hyponyms and hypernym (adapted from [1])

6.3.2 Taxonomy

With the increasing use of natural language processing techniques to improve explainability, semantic analysis of text resources has become a key challenge. Taxonomy construction aims to organize and classify entities according to their attributes and relations, which can simultaneously improve the efficiency of information retrieval and increase explainability. One of our previous research projects focuses on constructing a hierarchical taxonomy from the concepts.

A *taxonomy* is a partially ordered classification of entities according to their internal attributes and relations. A taxonomy is composed of hierarchical relationships (e.g., is-a or superclass-of relations) between entities. In linguistics terminology, a *hyponym* is a subtype or subclass of a *hypernym*. An example is shown in Figure 6.1.

Constructing such taxonomies can largely contribute to the semantic analysis of data sources, e.g., text corpus. It is an important tool to organize and manage information, which could enable users to easily navigate to their concepts of interest and improve information retrieval efficiency. For example, Velardi et al. [95] designed a semi-automated methodology to facilitate the design of a domain taxonomy and used the taxonomy to improve the accessibility of knowledge and data repositories in a Web community. Meanwhile, hierarchical relationships also commonly appear in domain modeling such as inheritance and a use case can be identifying super-classes and sub-classes from textual domain description for modeling purposes.

Our previous work proposes a general framework for automatically constructing a taxonomy using a language model and soft prompting. The general workflow has three steps: (1) *concept extraction* from the text corpus, (2) *hierarchical relation prediction* between concept pairs using language models with probing and soft prompt tuning [96] as a supervised learning task, and (3) *tree taxonomy construction* with hierarchical concept pairs using maximum spanning tree algorithm.

Generally, there are two categories of approaches for taxonomy construction, *manual approach* and *automatic approach*. In the manual approach, both construction and maintenance of the taxonomy rely on domain experts to decide the choice of the terminology and supervise the construction process [97]. Relying on human decisions makes this approach precise, but also makes it time-consuming, expensive, and unable to scale [98].

Due to the disadvantages of the manual approach, recent research mostly focuses on the automatic approach, which is large-scale and uses various natural language processing and machine learning techniques. Existing automatic taxonomy construction methods are classified into two categories: (1) pattern-based methods and (2) embedding-based methods.

Pattern-based methods construct taxonomies by using pre-defined lexico-syntactic patterns to extract hyponymy lexical pairs from the text corpus and then build the taxonomy tree. Hearst [99] is the pioneer of this approach, who identified a set of such patterns for the automatic acquisition of the hyponymy lexical relation from unrestricted text. An example of such patterns is a "such as" pattern, i.e., "NP_0 such as NP_1, NP_2...,NP_N" implies that for all NP_i, $1 \leq i \leq n$, NP_i is a subclass of NP_0. One clear limitation of Hearst's work is that both terms are required to appear in the same sentence. Inspired by previous work, Snow et al. [100] designed a system to automatically extract large numbers of useful lexico-syntactic patterns by using examples of known hypernym pairs. Besides, Nakashole et al. [101] proposed an efficient algorithm for mining textual patterns and built a subsumption taxonomy for a large lexical resource of such textual patterns. Recently, one participant in the SemEval-2018 Task 9 also presented an approach for automatic extraction of hypernymy relations from a corpus by using dependency patterns [102]. The commonality of

these pattern-based methods is that they rely on a large text corpus and integrate extensive linguistic knowledge. Secondly, natural language is consistently evolving and pattern-based methods may fail when expressions of parent-child relations change [103].

Embedding-based methods firstly learn an embedding representation of terms from text corpus or nodes from knowledge graphs and use embeddings to predict the is-a relation. For example, another participant to the SemEval-2018 Task 9 first used the skip-gram model to acquire word embeddings and then predict hypernyms for a given word based on cosine similarity scores [104]. Many other works [105,106] also employed word embeddings to attack this problem. However, it is argumentative whether these methods could learn hierarchy information, and the authors [107] also claimed that they do not learn a relation between two words.

With the help of embedding representations, it is possible to calculate various types of distance. Therefore, hierarchical clustering is also frequently adopted for taxonomy construction [103,108,109]. For example, Zhang et al. [103] used term embeddings and hierarchical clustering to recursively construct a taxonomy from a text corpus, whereas Martel et al. [108] construct a taxonomy with knowledge graph embeddings and hierarchical clustering. However, the problem is that hierarchical clustering can only separate entities into several clusters but it could not reveal the is-a relation between entities. Specifically, the result of hierarchical clustering is usually a tree where all input entities are assigned to different leaf nodes. Thus, non-leaf nodes have no assigned entities and one cannot identify the parent-child relationship by only observing the tree. Some addressed this problem by designing complex heuristic functions to decide how to assign entities to non-leaf nodes [103,108]. While the above-mentioned methods are based on Euclidean distance, some attempted hyperbolic embedding models (non-Euclidean geometries) and claimed that hyperbolic embeddings could be more efficient in learning hierarchy information [110–112].

Following the advent of LLMs, Chen et al. also attempt to leverage LLMs for taxonomy construction [113]. They systematically compare the effectiveness of using LLMs for taxonomy construction at a novel computer science taxonomy dataset.

6.3.3 Named Entity Recognition (NER)

Named entity recognition is one of the primary tasks in information extraction. It aims to automatically identify and classify text entities into predefined categories [44], so that machines or humans can better understand the meaning of the text, and ultimately extract useful information from documents. This process can be time-consuming and error-prone if done manually. With high-quality NER tools, the performance of many NLP applications can be further improved, such as information retrieval, question answering, and machine translation [114].

Various approaches have been proposed to achieve NER and can generally be classified into two categories, rule-based approaches and learning-based approaches [115]. Rule-based approaches use a part-of-speech tagging process to identify the type of words and explore contextual features rules [116]. This type of approach generally does not require annotated corpus resources for training but is hard to transfer to other domains.

Learning-based approaches are explored to detect entities and classify entities into the correct categories, including support vector machines (SVM), conditional random fields (CRF), and various large language models (LLMs) [117]. These approaches generally require learning an embedding representation of the input text for the final classification. Giorgi et al. [118] explore the effectiveness of using pre-trained transformer-based language models and BioBERT for the task of NER. They conclude that their pre-trained language models can identify features of entities in their dataset, CoNLL04 [119], which contains approximately two entities per sentence, and are able to exploit this pattern to get near-perfect performance on the majority of sentences in the corpus.

Our previous research explored the effectiveness of various LLMs for the task of NER. Our investigation on LLMs can also be further divided into encode-only models and decoder-only models. BERT-based models [26] are selected as the encode-only models and GPT-2 models proposed by OpenAI [27, 60, 120] are employed as the decoder-only models. Both types of models have similar basic mechanisms for converting natural language words into embedding representations and then classifying them.

Another early attempt by the author at automated domain model assessment involves using NER to extract entities from the natural language modeling problem text and assess the overlap between these entities and model elements, particularly class names. However, this approach works partially for class names and struggles to accurately assess relationships, especially multiplicities. As a result, this thesis does not follow the above-mentioned idea and instead focuses on matching model elements between a candidate model and a reference model. Additionally, since most NER approaches convert entities into text embeddings before classifying them into predefined labels, this thesis also adopts similar embedding techniques to obtain vector representations of model elements to find element matches.

6.4 Summary

In summary, this chapter first examines the related work on automated domain model evaluation, as outlined in Section 6.1. Herein, we comprehensively explore various existing approaches, explaining their intricacies and distinguishing them from our proposed approach. Subsequently, we present related NLP research applied to model-driven engineering, as demonstrated in Section 6.2. The first two subsections of Section 6.2 examine automated domain modeling and goal modeling, while the last subsection highlights applying LLMs for MDE. Furthermore, we shed light on additional research endeavors undertaken throughout the course of my master's academic journey, detailed in Section 6.3. This experience has further inspired our proposed approach in this thesis of applying embedding for automated domain model assessment. In the next chapter, we will synthesize the findings and formulate a concluding statement for this thesis. We will also discuss potential future directions.

Chapter 7

Conclusion

This chapter summarizes the primary contributions and findings of this thesis in Section 7.1, through the exploration of the proposed research questions. Furthermore, this thesis is concluded by discussing potential avenues for future research in Section 7.2.

7.1 Contributions and Findings

This thesis introduces a novel algorithm for fully automated domain model assessment utilizing embeddings and graph comparison techniques. The algorithm automatically matches model elements between a candidate model and a reference model in textual representation, subsequently generating a grade for the candidate model without human intervention. This thesis evaluates the algorithm using a dataset comprising 20 real-world student submissions from a university undergraduate-level modeling course. A reusable evaluation pipeline is designed to compare the element matches produced by the algorithm with those generated by a human grader.

The following paragraphs review the findings from addressing each research question.

RQ 1

What is the performance of the algorithm in matching a candidate domain model to a reference model regarding classes, attributes, and relationships?

RQ 1 aims to evaluate the performance of the algorithm in matching model elements between two domain models. RQ 1 also evaluates the performance of the algorithm in matching classes, attributes, and relationships using precision, recall, and F_1-scores. The algorithm achieves an average precision of 0.7425, recall of 0.9332, and F_1-score of 0.8239 for class matching. Additionally, it achieves average precision, recall, and F_1-score values of 0.7274, 0.7861, and 0.7456, respectively, for attribute matching, and 0.8556, 0.7628, and 0.7958, respectively, for relationship matching. These results indicate that although the proposed algorithm exhibits impressive performance in matching model elements, there remains potential for enhancement in its performance. While the domain elements matched by the algorithm are generally correct, there are instances of mismatches between elements. Additionally, the algorithm encounters the greatest difficulty in matching attributes, compared to classes and relationships.

RQ 2

To what extent does the algorithm-generated grade compare with those produced by human grading?

RQ 2 aims to investigate whether the statistics generated by the algorithm can approximate those from other approaches including manual grading. An internal comparison is conducted wherein the precision, recall, and F_1-scores generated by the algorithm are compared with those derived from human grading by the author of this thesis. The Mean Absolute Error (MAE) for each metric is small, all below 0.05. Specifically, the MAE for class precision is 0.04923, for class recall it is 0.04130, and for class F_1-score it is 0.4507. For attribute precision, the MAE is 0.04723, for attribute recall it is 0.04712, and for attribute F_1-score it is 0.04427. Regarding relationship metrics, the MAE for relationship precision is 0.04110, for relationship recall it is 0.2891, and for relationship F_1-score it is 0.03397. Notably, the MAE between grades is only 0.03096. To further validate the algorithm's results, statistical tests are performed. The analysis reveals no statistically significant differences in the precision, recall, and F_1-scores for attribute matching when comparing the algorithm's output

to the author's manual grading. However, statistically significant differences are identified in the precision, recall, and F_1-scores related to class matching and relationship matching. Such a statistically significant difference also extended to the final grades generated by the algorithm, as contrasted with the author's grading.

RQ 2 additionally conducts an external comparison, wherein it compares the grades generated by the algorithm with those from an existing benchmark. The MAE for numerical grades is similarly small, at 0.0456, and the MAE for letter grades is exactly 1. The algorithm-generated numerical grades also exhibit no statistically significant difference when compared to the external benchmark. The results illustrate that the algorithm is capable of generating meaningful precision, recall, F_1-scores, and grades that closely approximate the ground truth values for assessing a domain model. However, further improvement is necessary for the algorithm to fully substitute human grading in real-world applications, such as in educational scenarios.

Conclusion Statement

This thesis has significantly contributed to the research area of automated assessment of domain models against reference models. It innovatively integrates word embeddings, sentence embeddings, and graph comparison techniques to match two domain models and ultimately provides insightful assessments.

Moreover, this approach is available as a Python script, addressing the issue of technical debt found in some previous research attempts in the field. The primary goal of automated domain model assessment is to reduce the workload for researchers to manually evaluate domain models. It is counterproductive if researchers still have to spend considerable time installing and preparing the environment for the automated assessment approaches. Furthermore, this approach does not require input domain models to be in a rigid modeling language or specific file formats; it only requires them to be provided as natural language text, thereby enhancing adaptability.

This work impacts both the research and educational communities. For instance, the algorithm can evaluate domain models generated by LLMs against reference models curated by modeling experts, reducing the time and effort needed for manual evaluation and providing researchers with immediate feedback on their modeling approaches. Although experiments indicate that the algorithm does not yet reach human graders' assessments in educational scenarios, it can still serve as a valuable starting point for grading. Additionally, students can use the algorithm as a starting point to review their work for practice when reference models are provided.

7.2 Opportunities for Future Research

Numerous promising avenues for future work can build upon the foundations laid in this thesis. One potential application of this algorithm is in educational settings, where instructors could employ it to streamline the grading process of student submissions. Currently, no graphical user interface (GUI) has been implemented. An exciting direction for future research would be to develop an end-to-end software solution capable of grading two domain models using the algorithm and utilize it in real-world educational scenarios. This software would require user interfaces to effectively manage the inputs and demonstrate the algorithm's outputs. Additionally, the algorithm-generated matches should be visually represented to provide more intuitive feedback to users.

The algorithm implements a multi-stage and cascading process to match model elements. One promising avenue for enhancement involves integrating human feedback into the loop. For instance, the algorithm initiates the matching process by dealing with enumeration classes, with subsequent matches influenced by these initial resulting matches. Consequently, incorporating human feedback into the matches produced at each stage holds the potential to improve subsequent matches and enhance overall performance.

Another avenue of potential improvement involves integrating the problem description directly into the algorithm. Currently, the algorithm operates independently of the modeling

problem description, which contains rich information about the problem domain. By incorporating this description into the algorithm, it can augment the information available for element matching. For example, the algorithm could extract sentences that specifically describe each model element and incorporate them into the element embeddings. This enriched context may potentially enhance the accuracy and relevance of the matching process.

Appendix A

Modeling Problem Description and Reference Model

A.1 Problem Description

A smart home automation system (SHAS) allows various users to automatically manage smart home automation tasks. A smart home (located at a physical address) consists of several rooms, each of which may contain sensor devices and actuator (controller) devices of different types (e.g., temperature sensor, movement sensor, light controller, lock controller). Each sensor and actuator has a unique device identifier. Once a new sensor or actuator is activated or deactivated, SHAS will recognize the change and update its infrastructure map. When SHAS is operational, a sensor device periodically provides sensor readings (recording the measured value and the timestamp). Similarly, a predefined set of control commands (e.g., lockDoor, turnOnHeating) can be sent to the actuator devices with the timestamp and the status of the command (e.g., requested, completed, failed, etc.). All sensor readings and control commands for a smart home are recorded by SHAS in an activity log. Relevant alerts in a smart home can be set up and managed by its owner by setting up automation rules. An automation rule has a precondition and an action. The precondition is a Boolean expression constructed from relational terms connected by basic Boolean operators (AND, OR, NOT). Atomic relational terms may refer to rooms, sensors, actuators, sensor readings,

and control commands. The action is a sequence of control commands. For example, a sample rule could specify: when actualTemperature by Device #1244 in Living Room < 18 and window is closed then turnOnHeating in Living Room. Automation rules can be created, edited, activated, and deactivated by owners. Only deactivated rules can be edited. Rules can also depend on or conflict with other rules, thus a complex rule hierarchy can be designed. SHAS records whenever an active rule is triggered using a timestamp.

A.2 Reference Domain Model

The class diagram of the reference domain model to the modeling problem in Appendix A.1 is shown in Figure A.1. It is created by a modeling expert. Additionally, the textual representation of the domain model in our EBNF format is demonstrated in Listing A.1.

```
Enumerations:
DeviceStatus(activated, deactivated)
CommandType(lockDoor, turnOnHeating)
CommandStatus(requested, completed, failed)
RuleStatus(created, edited, activated, deactivated)
BinaryOp(AND, OR)

Classes:
SHAS()
SmartHome()
User(string name)
Address(string city, string postalCode, string street, string aptNumber)
Room()
abstract Device(DeviceStatus deviceStatus, int deviceID)
SensorDevice()
ActuatorDevice()
ActivityLog()
abstract RuntimeElement(time timestamp)
SensorReading(double value)
ControlCommand (CommandType commandType, CommandStatus commandStatus)
AlertRule (RuleStatus ruleStatus)
abstract BooleanExpression()
RelationalTerm()
NotExpression()
BinaryExpression(BinaryOp binaryOp)
CommandSequence()

Relationships:
1 SHAS contain * SmartHome
1 SHAS contain * User
1 SmartHome contain 0..1 Address
1 SmartHome contain * Room
1 SmartHome contain 0..1 ActivityLog
* SmartHome associate * User
1 SmartHome contain * AlertRule
1 Room contain * SensorDevice
1 Room contain * ActuatorDevice
1 ActivityLog contain * SensorReading
1 ActivityLog contain * ControlCommand
* SensorReading associate 1 SensorDevice
* ControlCommand associate 1 ActuatorDevice
1 AlertRule contain 0..1 BooleanExpression
1 AlertRule contain * CommandSequence
* RelationalTerm associate 0..1 Room
* RelationalTerm associate 0..1 SensorDevice
* RelationalTerm associate 0..1 ActuatorDevice
* RelationalTerm associate 0..1 SensorReading
* RelationalTerm associate 0..1 ControlCommand
0..1 NotExpression associate 1 BooleanExpression
0..1 BinaryExpression associate 1 BooleanExpression
0..1 BinaryExpression associate 1 BooleanExpression
* CommandSequence associate 0..1 CommandSequence
1 CommandSequence contain 0..1 ControlCommand
SensorReading inherit RuntimeElement
```

```
ControlCommand inherit RuntimeElement
NotExpression inherit BooleanExpression
BinaryExpression inherit BooleanExpression
RelationalTerm inherit BooleanExpression
SensorDevice inherit Device
ActuatorDevice inherit Device
```

Listing A.1: Reference domain model in EBNF format

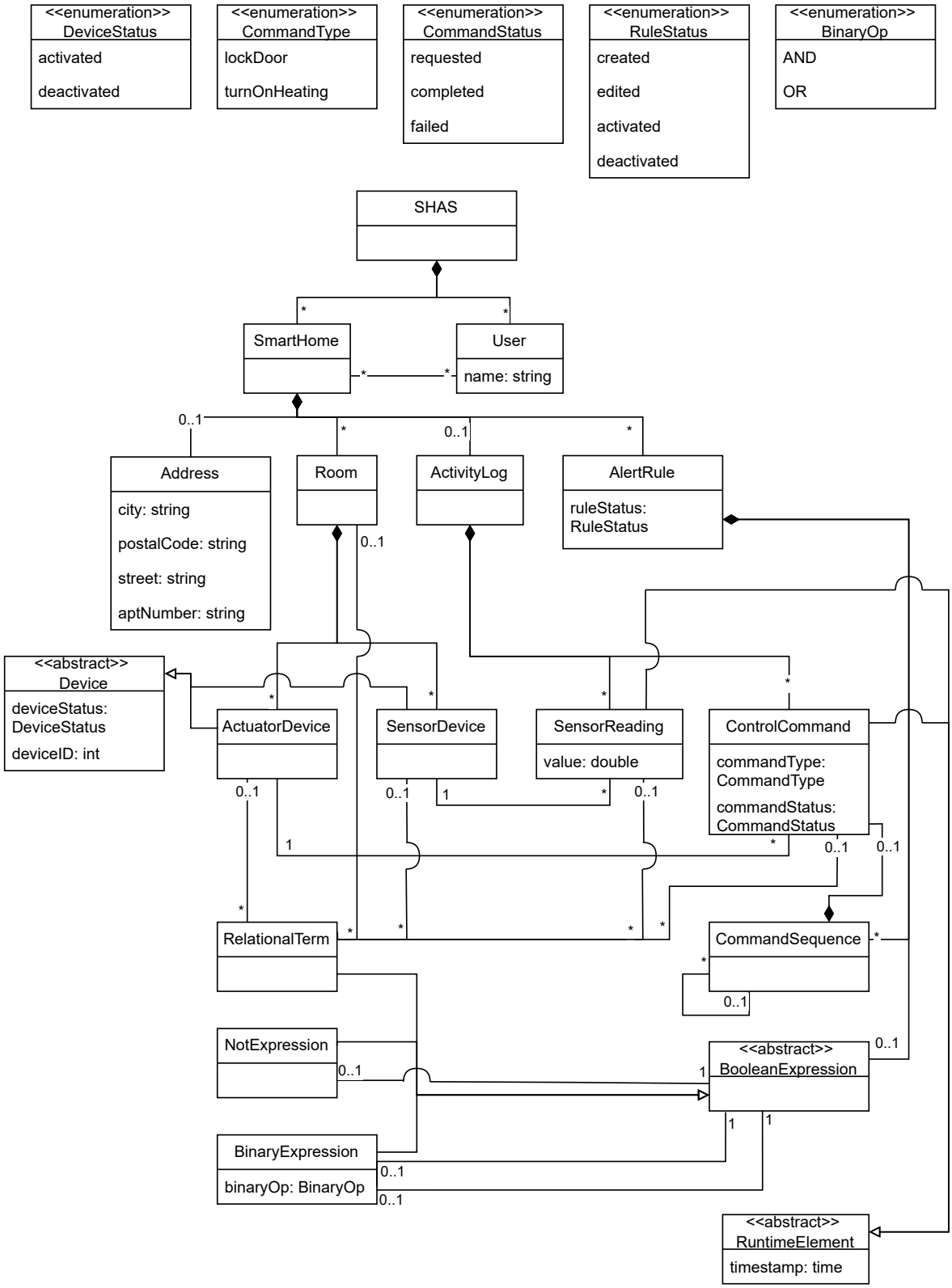


Figure A.1: Class diagram of the reference domain model

Bibliography

[1] Y. Verma, "A complete guide to using wordnet in nlp applications." https://analyticsindiamag.com/a-complete-guide-to-using-wordnet-in-nlp-applications/, 2021 (accessed December 7, 2023).

[2] K. Chen, Y. Yang, B. Chen, J. A. H. López, G. Mussbacher, and D. Varró, "Automated domain modeling with large language models: A comparative study," in *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 162–172, IEEE, 2023.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[4] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[6] M. B. Chaaben, L. Burgueño, and H. Sahraoui, "Towards using few-shot prompt learning for automating model completion," in *Proceedings of the 45th International Con-*

ference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '23, p. 7–12, IEEE Press, 2023.

[7] B. Chen, K. Chen, S. Hassani, Y. Yang, D. Amyot, L. Lessard, G. Mussbacher, M. Sabetzadeh, and D. Varró, "On the use of gpt-4 for creating goal models: an exploratory study," in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, pp. 262–271, IEEE, 2023.

[8] P. Singh, Y. Boubekeur, and G. Mussbacher, "Detecting mistakes in a domain model," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, (New York, NY, USA), p. 257–266, Association for Computing Machinery, 2022.

[9] Y. Boubekeur, G. Mussbacher, and S. McIntosh, "Automatic assessment of students' software models using a simple heuristic and machine learning," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[10] S. Yang and H. Sahraoui, "Towards automatically extracting UML class diagrams from natural language specifications," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, (New York, NY, USA), p. 396–403, Association for Computing Machinery, 2022.

[11] W. Bian, O. Alam, and J. Kienzle, "Automated grading of class diagrams," in *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems*, MODELS '19, p. 700–709, IEEE Press, 2021.

[12] A. Jayal and M. Shepperd, "The problem of labels in e-assessment of diagrams," *Journal on Educational Resources in Computing (JERIC)*, vol. 8, no. 4, pp. 1–13, 2009.

[13] R. Hasker, "Umlgrader: an automated class diagram grader," *Journal of Computing Sciences in Colleges*, vol. 27, pp. 47–54, 10 2011.

[14] C. Tselonis, J. Sargeant, and M. McGee Wood, "Diagram matching for human-computer collaborative assessment," in *Proceedings of the 9th International Computer Assisted Assessment Conference*, pp. 1–15, Loughborough University, May 2005.

[15] L. Auxepaules, D. Py, and T. Lemeunier, "A diagnosis method that matches class diagrams in a learning environment for object-oriented modeling," in *Proceedings of the 2008 Eighth IEEE International Conference on Advanced Learning Technologies*, ICALT '08, (USA), p. 26–30, IEEE Computer Society, 2008.

[16] M. A. Garzón, H. Aljamaan, and T. C. Lethbridge, "Umple: A framework for model driven development of object-oriented systems," in *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (saner)*, pp. 494–498, IEEE, 2015.

[17] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[18] B. Chen, K. Chen, Y. Yang, A. Amini, B. Saxena, C. Chávez-García, M. Babaei, A. Feizpour, and D. Varró, "Towards improving the explainability of text-based information retrieval with knowledge graphs," *arXiv preprint arXiv:2301.06974*, 2023.

[19] R. S. Scowen, "Generic base standards," in *Proceedings 1993 Software Engineering Standards Symposium*, pp. 25–34, IEEE, 1993.

[20] J. Barnard, "What is embedding?." `https://www.ibm.com/topics/embedding`, Dec 2023. Accessed on 2024-02-13.

[21] F. Almeida and G. Xexéo, "Word embeddings: A survey," *arXiv preprint arXiv:1901.09069*, 2019.

[22] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural*

Language Processing (EMNLP) (A. Moschitti, B. Pang, and W. Daelemans, eds.), (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, (Red Hook, NY, USA), p. 3111–3119, Curran Associates Inc., 2013.

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[25] J. A. H. López, C. Durá, and J. S. Cuadrado, "Word embeddings for model-driven engineering," in *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 151–161, IEEE, 2023.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[28] J. Han, M. Kamber, and J. Pei, "2 - getting to know your data," in *Data Mining (Third Edition)* (J. Han, M. Kamber, and J. Pei, eds.), The Morgan Kaufmann Series in Data Management Systems, pp. 39–82, Boston: Morgan Kaufmann, third edition ed., 2012.

[29] R. Sedgewick, *Algorithms in java, part 5: Graph algorithms.* Addison-Wesley Professional, 2003.

[30] M. Grohe and P. Schweitzer, "The graph isomorphism problem," *Commun. ACM*, vol. 63, p. 128–134, Oct 2020.

[31] X. Chen, H. Huo, J. Huan, and J. S. Vitter, "An efficient algorithm for graph edit distance computation," *Knowledge-Based Systems*, vol. 163, pp. 762–775, 2019.

[32] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[33] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proceedings of the International Conference on Pattern Recognition Applications and Methods - Volume 1*, ICPRAM 2015, (Setubal, PRT), p. 271–278, SCITEPRESS - Science and Technology Publications, Lda, 2015.

[34] M. G. Kendall, "The treatment of ties in ranking problems," *Biometrika*, vol. 33, no. 3, pp. 239–251, 1945.

[35] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[36] A. Ross, V. L. Willson, A. Ross, and V. L. Willson, "Paired samples t-test," *Basic and Advanced Statistical Tests: Writing Results Sections and Creating Tables and Figures*, pp. 17–19, 2017.

[37] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[38] R. Saini, G. Mussbacher, J. L. C. Guo, and J. Kienzle, "Machine learning-based incremental learning in interactive domain modelling," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, p. 176–186, ACM, 2022.

[39] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature," *Geoscientific Model Development*, vol. 7, no. 3, pp. 1247–1250, 2014.

[40] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.

[41] G. Liu, C. Wang, H. Qin, J. Fu, and Q. Shen, "A novel hybrid machine learning model for wind speed probabilistic forecasting," *Energies*, vol. 15, no. 19, p. 6942, 2022.

[42] V. Bewick, L. Cheek, and J. Ball, "Statistics review 7: Correlation and regression," *Critical care*, vol. 7, pp. 1–9, 2003.

[43] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, Soviet Union, 1966.

[44] J. Santoso, E. I. Setiawan, C. N. Purwanto, E. M. Yuniarno, M. Hariadi, and M. H. Purnomo, "Named entity recognition for extracting concept in ontology building on indonesian language using end-to-end bidirectional long short term memory," *Expert Systems with Applications*, vol. 176, p. 114856, 2021.

[45] M. Gala, "Unified modeling language (UML) generation from user requirements in natural language." https://uu.diva-portal.org/smash/get/diva2:1809096/FULLTEXT01.pdf, 2023.

[46] A. Chiche and B. Yitagesu, "Part of speech tagging: a systematic review of deep learning and machine learning approaches," *Journal of Big Data*, vol. 9, no. 1, pp. 1–25, 2022.

[47] M. Robeer, G. Lucassen, J. M. van der Werf, F. Dalpiaz, and S. Brinkkemper, "Automated extraction of conceptual models from user stories via nlp," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, (Los Alamitos, CA, USA), pp. 196–205, IEEE Computer Society, September 2016.

[48] H. Herchi and W. B. Abdessalem, "From user requirements to UML class diagram," *arXiv preprint arXiv:1211.0713*, 2012.

[49] T. Güneş and F. B. Aydemir, "Automated goal model extraction from user stories using NLP," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pp. 382–387, IEEE, 2020.

[50] C. Wu, C. Wang, T. Li, and Y. Zhai, "A node-merging based approach for generating iStar models from user stories," *Software Engineering and Knowledge Engineering*, pp. 257–262, 2022.

[51] J. Franců and P. Hnětynka, "Automated generation of implementation from textual system requirements," in *Software Engineering Techniques: Third IFIP TC 2 Central and East European Conference, CEE-SET 2008, Brno, Czech Republic, October 13-15, 2008, Revised Selected Papers 3*, pp. 34–47, Springer, 2011.

[52] S. Amdouni, W. B. A. Karaa, and S. Bouabid, "Semantic annotation of requirements for automatic uml class diagram generation," *arXiv preprint arXiv:1107.3297*, 2011.

[53] L. Burgueño, R. Clarisó, S. Gérard, S. Li, and J. Cabot, "An NLP-based architecture for the autocompletion of partial domain models," in *Advanced Information Systems Engineering: 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28–July 2, 2021, Proceedings*, pp. 91–106, Springer, 2021.

[54] M. Weyssow, H. Sahraoui, and E. Syriani, "Recommending metamodel concepts during modeling activities with pre-trained language models," *Software and Systems Modeling*, vol. 21, no. 3, pp. 1071–1089, 2022.

[55] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, "On the assessment of generative ai in modeling tasks: An experience report with chatgpt and uml," *Softw. Syst. Model.*, vol. 22, p. 781–793, May 2023.

[56] B. Combemale, J. Gray, and B. Rumpe, "Chatgpt in software modeling," *Softw. Syst. Model.*, vol. 22, p. 777–779, May 2023.

[57] Q. Zhou, T. Li, and Y. Wang, "Assisting in requirements goal modeling: a hybrid approach based on machine learning and logical reasoning," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, (New York, NY, USA), p. 199–209, Association for Computing Machinery, 2022.

[58] E. Yu, P. Giorgini, N. Maiden, J. Mylopoulos, and S. Fickas, "Modeling strategic relationships for process reengineering," in *Social Modeling for Requirements Engineering*, pp. 66–87, The MIT Press, 2011.

[59] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heinz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *arXiv preprint arXiv:2111.01243*, 2021.

[60] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[61] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[62] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[63] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[64] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural

language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 7871–7880, Association for Computational Linguistics, 2020.

[65] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 6000–6010, Curran Associates Inc., 2017.

[66] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training." `https://openai.com/research/language-unsupervised`, 2018.

[67] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[68] B. AlKhamissi, M. Li, A. Celikyilmaz, M. Diab, and M. Ghazvininejad, "A review on language models as knowledge bases," *arXiv preprint arXiv:2204.06031*, 2022.

[69] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023.

[70] H.-G. Fill, P. Fettke, and J. Köpke, "Conceptual modeling and large language models: impressions from first experiments with ChatGPT," *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 18, pp. 1–15, 2023.

[71] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd ed., 2009.

[72] E. H. Y. Lim, J. N. K. Liu, and R. S. T. Lee, *Text Information Retrieval*, ch. Text Information Retrieval, pp. 27–36. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[73] M. R. Islam, M. U. Ahmed, S. Barua, and S. Begum, "A systematic review of explainable artificial intelligence in terms of different application domains and tasks," *Applied Sciences*, vol. 12, no. 3, p. 1353, 2022.

[74] I. Chios and S. Verberne, "Helping results assessment by adding explainable elements to the deep relevance matching model," *arXiv preprint arXiv:2106.05147*, 2021.

[75] J. Ramos and C. Eickhoff, "Explainability in transparent information retrieval systems." https://api.semanticscholar.org/CorpusID:147702637, 2019. Accessed on 2024-01-19.

[76] S. Polley, A. Janki, M. Thiel, J. Hoebel-Mueller, and A. Nuernberger, "Exdocs: Evidence-based explainable document search," in *Proceedings of the ACM SIGIR Workshop on Causality in Search and Recommendation*, vol. 2911, pp. 1–7, Association for Computing Machinery, 2021.

[77] J. Singh and A. Anand, "Exs: Explainable search using local model agnostic interpretability," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, (New York, NY, USA), p. 770–773, Association for Computing Machinery, 2019.

[78] M. A. Qureshi and D. Greene, "Eve: explainable vector based embedding technique using wikipedia," *Journal of Intelligent Information Systems*, vol. 53, no. 1, pp. 137–165, 2019.

[79] A. Panigrahi, H. V. Simhadri, and C. Bhattacharyya, "Word2Sense: Sparse interpretable word embeddings," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (A. Korhonen, D. Traum, and L. Màrquez, eds.), (Florence, Italy), pp. 5692–5705, Association for Computational Linguistics, July 2019.

[80] R. Reinanda, E. Meij, M. de Rijke, *et al.*, "Knowledge graphs: An information retrieval perspective," *Foundations and Trends® in Information Retrieval*, vol. 14, no. 4, pp. 289–444, 2020.

[81] I. Tiddi and S. Schlobach, "Knowledge graphs as tools for explainable machine learning: A survey," *Artificial Intelligence*, vol. 302, p. 103627, 2022.

[82] Z. Yang, "Biomedical information retrieval incorporating knowledge graph for explainable precision medicine," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, (New York, NY, USA), pp. 2486–2486, Association for Computing Machinery, 2020.

[83] H. Abu-Rasheed, C. Weber, J. Zenkert, M. Dornhöfer, and M. Fathi, "Transferrable framework based on knowledge graphs for generating explainable results in domain-specific, intelligent information retrieval," *Informatics*, vol. 9, no. 1, p. 6, 2022.

[84] S. Balaneshinkordan and A. Kotov, "An empirical comparison of term association and knowledge graphs for query expansion," in *European conference on information retrieval*, pp. 761–767, Springer, 2016.

[85] C. Xiong and J. Callan, "Esdrank: Connecting query and documents through external semi-structured data," in *Proceedings of the 24th ACM international on conference on information and knowledge management*, (New York, NY, USA), pp. 951–960, Association for Computing Machinery, 2015.

[86] F. Ensan and E. Bagheri, "Document retrieval model through semantic linking," in *Proceedings of the tenth ACM international conference on web search and data mining*, (New York, NY, USA), pp. 181–190, Association for Computing Machinery, 2017.

[87] C. Xiong, R. Power, and J. Callan, "Explicit semantic ranking for academic search via knowledge graph embedding," in *Proceedings of the 26th international conference on world wide web*, (Republic and Canton of Geneva, CHE), pp. 1271–1279, International World Wide Web Conferences Steering Committee, 2017.

[88] Z. Liu, C. Xiong, M. Sun, and Z. Liu, "Entity-duet neural ranking: Understanding the role of knowledge graph semantics in neural information retrieval," *arXiv preprint arXiv:1805.07591*, 2018.

[89] C. Xiong, J. Callan, and T.-Y. Liu, "Word-entity duet representations for document ranking," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, (New York, NY, USA), p. 763–772, Association for Computing Machinery, 2017.

[90] Y. Xian, Z. Fu, S. Muthukrishnan, G. de Melo, and Y. Zhang, "Reinforcement knowledge graph reasoning for explainable recommendation," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, (New York, NY, USA), p. 285–294, Association for Computing Machinery, 2019.

[91] P. Ferragina and U. Scaiella, "Fast and accurate annotation of short texts with wikipedia pages," *IEEE Software*, vol. 29, no. 1, pp. 70–75, 2011.

[92] F. Piccinno and P. Ferragina, "From tagme to wat: A new entity annotator," in *Proceedings of the First International Workshop on Entity Recognition and Disambiguation*, ERD '14, (New York, NY, USA), p. 55–62, Association for Computing Machinery, 2014.

[93] Y. Yang, W.-t. Yih, and C. Meek, "WikiQA: A challenge dataset for open-domain question answering," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (L. Màrquez, C. Callison-Burch, and J. Su, eds.), (Lisbon, Portugal), pp. 2013–2018, Association for Computational Linguistics, Sept. 2015.

[94] E. M. Voorhees, "The trec robust retrieval track," *SIGIR Forum*, vol. 39, p. 11–20, June 2005.

[95] P. Velardi, A. Cucchiarelli, and M. Petit, "A taxonomy learning method and its application to characterize a scientific web community," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 2, pp. 180–191, 2007.

[96] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," *arXiv preprint arXiv:2104.08691*, 2021.

[97] R. Sujatha, R. Bandaru, and R. Rao, "Taxonomy construction techniques–issues and challenges," *Indian Journal of Computer Science and Engineering*, vol. 2, no. 5, pp. 661–671, 2011.

[98] M. Centelles, "Taxonomies for categorisation and organisation in websites." https://arxiu-web.upf.edu/hipertextnet/en/numero-3/taxonomias.html, 2005. Accessed on 2024-01-25.

[99] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in *Proceedings of the 14th Conference on Computational Linguistics - Volume 2*, COLING '92, (USA), p. 539–545, Association for Computational Linguistics, 1992.

[100] R. Snow, D. Jurafsky, and A. Y. Ng, "Learning syntactic patterns for automatic hypernym discovery," in *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, (Cambridge, MA, USA), p. 1297–1304, MIT Press, 2004.

[101] N. Nakashole, G. Weikum, and F. Suchanek, "Patty: a taxonomy of relational patterns with semantic types," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, (USA), p. 1135–1145, Association for Computational Linguistics, 2012.

[102] M. Onofrei, I. Hulub, D. Trandabăț, and D. Gîfu, "Apollo at SemEval-2018 task 9: Detecting hypernymy relations using syntactic dependencies," in *Proceedings of the 12th International Workshop on Semantic Evaluation* (M. Apidianaki, S. M. Mohammad, J. May, E. Shutova, S. Bethard, and M. Carpuat, eds.), (New Orleans, Louisiana), pp. 898–902, Association for Computational Linguistics, June 2018.

[103] C. Zhang, F. Tao, X. Chen, J. Shen, M. Jiang, B. Sadler, M. Vanni, and J. Han, "Taxogen: Unsupervised topic taxonomy construction by adaptive term embedding and clustering," in *Proceedings of the 24th ACM SIGKDD International Conference on*

Knowledge Discovery & Data Mining, KDD '18, (New York, NY, USA), p. 2701–2709, Association for Computing Machinery, 2018.

[104] A. Maldonado and F. Klubička, "Adapt at semeval-2018 task 9: Skip-gram word embeddings for unsupervised hypernym discovery in specialised corpora," in *Proceedings of The 12th International Workshop on Semantic Evaluation*, (New Orleans, Louisiana), pp. 924–927, Association for Computational Linguistics, 2018.

[105] R. Fu, J. Guo, B. Qin, W. Che, H. Wang, and T. Liu, "Learning semantic hierarchies via word embeddings," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Baltimore, Maryland), pp. 1199–1209, Association for Computational Linguistics, June 2014.

[106] A. T. Luu, Y. Tay, S. C. Hui, and S. K. Ng, "Learning term embeddings for taxonomic relation identification using dynamic weighting neural network," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, (Austin, Texas), pp. 403–413, Association for Computational Linguistics, Nov. 2016.

[107] O. Levy, S. Remus, C. Biemann, and I. Dagan, "Do supervised distributional methods really learn lexical inference relations?," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (Denver, Colorado), pp. 970–976, Association for Computational Linguistics, May–June 2015.

[108] F. Martel and A. Zouaq, "Taxonomy extraction using knowledge graph embeddings and hierarchical clustering," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, (New York, NY, USA), p. 836–844, Association for Computing Machinery, 2021.

[109] H. Bai, F. Z. Xing, E. Cambria, and W.-B. Huang, "Business taxonomy construction using concept-level hierarchical clustering," *arXiv preprint arXiv:1906.09694*, 2019.

[110] M. Nickel and D. Kiela, "Poincaré embeddings for learning hierarchical representations," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 6341–6350, Curran Associates Inc., 2017.

[111] R. Aly, S. Acharya, A. Ossa, A. Köhn, C. Biemann, and A. Panchenko, "Every child should have parents: a taxonomy refinement algorithm based on hyperbolic term embeddings," *arXiv preprint arXiv:1906.02002*, 2019.

[112] M. Le, S. Roller, L. Papaxanthos, D. Kiela, and M. Nickel, "Inferring concept hierarchies from text corpora via hyperbolic embeddings," *arXiv preprint arXiv:1902.00913*, 2019.

[113] B. Chen, F. Yi, and D. Varró, "Prompting or fine-tuning? a comparative study of large language models for taxonomy construction," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, (Los Alamitos, CA, USA), pp. 588–596, IEEE Computer Society, 2023.

[114] B. Mohit, "Named entity recognition," in *Natural language processing of semitic languages*, pp. 221–245, Springer, 2014.

[115] A. Goyal, V. Gupta, and M. Kumar, "Recent named entity recognition and classification techniques: a systematic review," *Computer Science Review*, vol. 29, pp. 21–43, 2018.

[116] R. Alfred, L. Leong, C. On, and P. Anthony, "Malay named entity recognition based on rule-based approach," *International Journal of Machine Learning and Computing*, vol. 4, pp. 300–306, 06 2014.

[117] M. Jiang, Y. Chen, M. Liu, S. T. Rosenbloom, S. Mani, J. C. Denny, and H. Xu, "A study of machine-learning-based approaches to extract clinical entities and their assertions from discharge summaries," *Journal of the American Medical Informatics Association*, vol. 18, no. 5, pp. 601–606, 2011.

[118] J. Giorgi, X. Wang, N. Sahar, W. Y. Shin, G. D. Bader, and B. Wang, "End-to-end named entity recognition and relation extraction using pre-trained language models," *arXiv preprint arXiv:1912.13415*, 2019.

[119] D. Roth and W.-t. Yih, "A linear programming formulation for global inference in natural language tasks," tech. rep., Illinois Univ at Urbana-Champaign Dept of Computer Science, 2004.

[120] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), pp. 1877–1901, Curran Associates Inc., 2020.