Embedding-based Automated Assessment of Domain Models

Kua Chen
McGill University
Montreal, Quebec, Canada

Boqi Chen
McGill University
Montreal, Quebec, Canada

Yujing Yang
McGill University
Montreal, Quebec, Canada

Gunter Mussbacher
McGill University
Montreal, Quebec, Canada

Dániel Varró*
Linköping University
Linköping, Sweden

ABSTRACT

Domain modeling is an essential component in many software engineering courses since it serves as a way to represent and understand the concepts and relationships in a problem domain. Course instructors evaluate student-generated diagrams manually, comparing them against a reference solution and providing feedback. However, as enrollment in software engineering courses continues to rise, manual grading of a large number of student submissions becomes an overwhelming and time-intensive task for instructors. Hence, there is a need for automated assessment of domain models which assists course instructors during the grading process. In this paper, we propose a novel text embedding-based approach that automatizes the assessment of domain models expressed in a textual domain-specific language, against reference solutions created by modeling experts. Our algorithm showcases remarkable proficiency in matching model elements across domain models, achieving an F_1-score of 0.82 for class matching, 0.75 for attribute matching, and 0.80 for relation matching. Our algorithm also yields grades highly correlated with human grader assessments, with correlations exceeding 0.8 and mean absolute errors below 0.05.

CCS CONCEPTS

• **Social and professional topics → Student assessment**; • **Software and its engineering → Designing software**.

KEYWORDS

Domain modeling, text embeddings, domain model assessment

ACM Reference Format:
Kua Chen, Boqi Chen, Yujing Yang, Gunter Mussbacher, and Dániel Varró. 2024. Embedding-based Automated Assessment of Domain Models. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3652620.3687774

*Also with McGill University.

Partially supported by the FRQNT-B2X project (file number: 319955), IT30340 Mitacs Accelerate, and the Wallenberg AI, Autonomous Systems and Software Program (WASP), Sweden.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687774

1 INTRODUCTION

Context and Motivation. Domain modeling is a core process in software engineering that builds a domain model in the form of a class diagram from various sources of information, such as documents and stakeholder interactions. Software engineering students usually learn domain modeling by interpreting textual problem descriptions and manually building a domain model by incrementally combining model elements. 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 attempt to identify the occurrence of model elements in the reference model within the student's model. As the number of students in software modeling courses increases, the grading becomes an overwhelming workload for course instructors.

Another motivation for automated domain model assessment is to facilitate research progress in domain modeling. Currently, most researchers need to manually evaluate the domain models generated from their proposed techniques. Such manual evaluation is time-consuming and highly dependent on human expertise. This problem becomes more challenging when large language models (LLMs) are used for automated domain modeling [7, 8]. Researchers still have to evaluate the generated domain model manually due to a lack of a proper automated assessment approach. This hinders the research progress in exploring the modeling ability of LLMs.

Objectives. In this paper, we aim to address the problem of fully automated domain model assessment. For that purpose, we propose an end-to-end domain model assessment algorithm, which does not require any human interaction or supervised training.

Contribution. Given two domain models, we present a novel approach for fully automated domain model assessment using text embeddings and graph comparison techniques. The specific contributions of this paper are the following:

- We formulate the domain model comparison problem as a model element matching problem and propose a fully automated model assessment pipeline
- We propose an automated match comparison workflow to assess the performance of our approach.
- We provide a new data set of 20 real student submissions for evaluating automated domain model assessment. The data set includes detailed matching information.
- We conduct an experiment using 20 real student solutions to analyze the precision, recall, and F_1-scores of our algorithm.

Added Value. Our novel algorithm aims at fully automating the assessment of domain models by matching model elements and deriving reliable grades for domain models, thus reducing the time and effort needed for human assessment. Our approach has the potential to make grading tasks and modeling research more efficient, providing a robust tool for both academic and industrial applications. Moreover, unlike in other machine learning approaches, the integrated text embeddings do not require training, while they can also handle more cases compared to rule-based approaches. Moreover, text embeddings can be easily substituted with more effective text embeddings, which has a direct impact on grading performance.

2 BACKGROUND

Domain Model Representation. In this paper, we focus on developing an algorithm built upon the domain model representation proposed in existing work on automated domain modeling [8]. The textual representation of two partial domain models from a smart home domain is shown in Table 1, including classes with attributes, enumerations with literals, and relations with multiplicities. These models will be used to explain the key concepts of our approach.

Table 1: Snippet of smart home domain models

Reference Model
Enumerations:
CommandStatus (Requested, Completed, Failed)
Classes:
ControlCommand (CommandStatus commandStatus)
ActuatorDevice ()
Relations:
0..* ControlCommand associate 1 ActuatorDevice
Candidate Model
Enumerations:
Status (Requested, Completed, Failed)
Classes:
ControlCommand (Status status)
Actuator ()
Relations:
0..* ControlCommand associate 1 Actuator

Word Embeddings. Word embeddings represent individual words in a fixed-dimension vector space so that computers can understand them. The word embedding used in this paper is trained using Skip-gram [15], which is a well-known word embedding method. Given a word, a neural network is trained to predict the surrounding words. The training objective of a Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence [15]. 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:

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

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 probability is often modeled using the softmax function:

$$p(w_O|w_I) = \frac{e^{u'^T_{w_O} \cdot u_{w_I}}}{\sum_{v=1}^{V} e^{u'^T_{w_v} \cdot v_{w_I}}} \tag{2}$$

where u and u' are vector representations of words. Skip-gram methods were pre-trained on general text sources like Wikipedia. Thus, when applying them to a specific domain, they may misinterpret some words. In 2023, Hernández et al. [14] applied Skip-gram methods with a large corpus of modeling texts and released WordE4MDE (Word Embeddings for MDE), which we use in this work.

Sentence Embeddings. Sentence embedding refers to the process of representing an entire sentence as a fixed-size vector. A very popular contextual sentence embedding is *pre-trained model embedding*, typically empowered by transformer models like BERT (Bidirectional Encoder Representations from Transformers) [13] and GPT (Generative Pre-trained Transformer) [16]. For the BERT model, the classification token is prepended to the input when a sequence of tokens is fed into it. These models are typically trained to ensure that semantically similar sentence pairs have embeddings that are close to each other and dissimilar pairs have embeddings that are far apart. The output representation of the classification token is used as the aggregated representation of the entire input sequence. For GPT models, the last token in the sequence is used as the sentence representation.

Cosine Similarity. Cosine similarity is measured by the cosine of the angle between two vectors and determines to what extent two vectors are pointing in the same direction [12]. Cosine similarity is often used in data analysis to measure two items' similarity. It is also applied in the model-driven engineering (MDE) field [19]. Given a properly trained embedding method, the embedding of texts that share similar meanings will have a larger cosine similarity value. Let us define n-dimensional vectors **A** and **B**. The cosine similarity is calculated:

$$\text{cosine similarity}(\mathbf{A}, \mathbf{B}) = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3}$$

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

Graph Similarity Measures. A simple graph is defined as a set of vertices V and a set of edges E that connect pairs of vertices [18]. Two graphs can have different ordering of vertices and edges but the same underlying structure, leading to the idea of graph isomorphism. Two graphs are isomorphic if there exists a function f from the vertices of G_1 to the vertices of G_2 (i.e., $f : V(G_1) \rightarrow V(G_2)$), such that: (1) f is a bijection, and (2) 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. Therefore, a widely used metric for graph similarity is the minimum graph edit distance (GED). The minimum 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 [9]. GED has been utilized for evaluating UML class diagrams [22]. Mathematically, the problem of finding the minimum GED can be defined as follows:

$$GED(g_1, g_2) = \min_{(e_1,...,e_k) \in \mathcal{P}(g_1,g_2)} \sum_{i=1}^{k} c(e_i) \tag{4}$$

where $\mathcal{P}(g_1, g_2)$ represents the sequence of edit paths transforming g_1 into an isomorphic graph as g_2, and $c(e)$ is the cost associated with each graph edit operation e.

3 METHOD

The essence of our approach (shown in Figure 1) is matching model elements between a candidate model and a (human) reference model to evaluate the quality of candidate models and then assign a grade calculated from statistical results of the matching. After a *Pre-processing Stage*, our approach consists of four main technical stages, *Stage 1* for *class matching*, *Stage 2* for *attribute matching*, *Stage 3* for *relation matching*, and *Stage 4* for computing *statistical results*. After *Stage 4*, the *Grading Stage* computes a final grade.

At the *Pre-processing Stage*, the algorithm takes two domain models represented in our domain-specific language and introduces nested hash maps to keep track of the status of each model element from the two models, including their matching information. As each following stage matches model elements between the candidate and reference models based on graph similarity measures, the nested hash maps eventually contain element matches and matching scores for calculating precision, recall, and F_1-score of each kind of model element (classes, attributes, and relations).

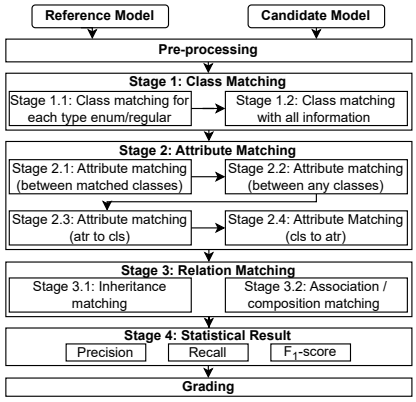

Figure 1: Overview of the proposed algorithm

3.1 Stage 1: Class Matching

Stage 1 focuses on matching classes between the candidate model and the reference model. This process starts with calculating the class embeddings. The embedding of a class is derived from the embeddings of its class name, attributes, and relations. The next step is calculating pair-wise cosine similarities between classes based on the class embeddings. Then the algorithm identifies matches between two sets of classes that minimize the cost of matches. In practice, to solve this set-matching problem, our algorithm converts domain models into two trivial graphs (graphs without edges). It computes the minimum GED from the candidate graph to the reference graph where the cost of each operation is derived from the similarity score between 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.

Stage 1.1: Class Matching within Types. *Stage 1.1* focuses on matching classes based on names and attributes within the same type. There are three types of classes: regular, abstract, and enumeration classes. In the MDE context, regular and abstract classes are conceptually similar. Therefore, we group regular and abstract classes into one matching pool and treat enumeration classes in another. In our running example from Table 1, [`CommandStatus`] and [`Status`] are grouped. Furthermore, [`ControlCommand`, `ActuatorDevice`] is grouped for matching with [`ControlCommand`,`Actuator`].

To compare classes, two types of embeddings are calculated: (1) Class-name-only and (2) Class-name-with-attributes. For the class-name-only embeddings, words in the name of the class are extracted. Then, the embedding is calculated as the average word embeddings. For the second type, attribute embeddings are calculated similarly to class names. The final embedding is the average of class name and attribute embeddings. Then, pair-wise similarities between classes can be calculated based on cosine similarity.

Cosine Similarities. We calculate 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. A weighted average is applied to the two types of embeddings to combine the similarity scores from class-name-only embeddings and class-name-with-attributes embeddings for aggregation.

Function `match_classes`. This function matches classes with cosine similarities and GED. It requires inputs including a list of reference class names, a list of candidate class names, a hash map where cosine similarities are stored, and a float parameter for threshold. At the beginning, 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 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. With a good embedding method, similar class pairs have high cosine similarities. Therefore, high-similarity vertex pairs have low cost and they are more likely to get matched. For instance, the cost of matching `ActuatorDevice` with `Actuator` is only 0.111, whereas matching with `ControlCommand` costs 0.631, hence `ActuatorDevice` will be matched with `Actuator`. The threshold parameter of 0.5 is used to filter some class pairs with low similarity values. If the similarity is lower than the threshold, the substitution cost will be adjusted to a large value. This prevents matching low-similarity class pairs. Overall, this function optimizes the vertex edit operation sequence transforming one graph to the other graph with the lowest cost. It eventually returns class matches like: [(`CommandStatus`, `Status`), (`ControlCommand`, `ControlCommand`), (`ActuatorDevice`, `Actuator`)]. Suppose there is another reference enumeration class called `CommandType`. Since there is no corresponding candidate class, it will not be matched, i.e., resulting in (`CommandType`, None).

Stage 1.2: Class Matching with All Information. *Stage 1.2* matches leftover classes in the candidate model and the reference model from the previous stage using all available information, including relations. For each leftover class, we gather its raw textual class representation, along with all associated relations. Sentence embeddings can capture the directionality in relations. Thus we employ sentence embedding techniques to transform this representation into a vector as the class embeddings. Next, we use the class embeddings to calculate the pair-wise cosine similarities between the reference and candidate classes. The same matching algorithm

is used to match these classes by using the pairwise similarity calculated from the sentence embedding. This function matches input classes and returns class matches.

Score Assignment. Our algorithm assigns a score to each element in the matches. For both *Stage 1.1* and *Stage 1.2*, the algorithm checks whether class matches possess the same type (enumeration, regular, or abstract) after finding them from the `match_classes` function. If both classes in a class match have the same type, they are assigned a score of one. Otherwise, the score is set to 0.5.

3.2 Stage 2: Attribute Matching

In *Stage 2*, the algorithm focuses on matching the attributes between the reference and candidate models. Leftover classes from previous stages are also considered for matching attributes in later stages. 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

Attribute matching follows a similar logic to class matching.

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*. Then, we use the `match_attributes` function to match attributes. For instance, `CommandStatus` and `Status` have been matched from *Stage 1*. Then, their enumeration literals are grouped for matching. Likewise, attributes in reference class `ControlCommand` and candidate class `ControlCommand` are considered for matching.

Function `match_attributes`. For each class match, the algorithm collects reference attributes and candidate attributes in two separate lists. Attributes are embedded for computing pair-wise cosine similarities. Subsequently, the two lists and the cosine similarities 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*. The `match_attributes` function also returns a list of attribute matches.

Score Assignment. Our algorithm assigns matching scores to each attribute in the returned matches. Each attribute includes a type, e.g., int, string, or enumeration class. Sometimes there could be an anti-pattern in the type. Therefore, our algorithm conducts *type checking* on matched attributes in each pair with three rules: (1) if any attribute type is a regular class or abstract class, mark them as partially correct, (2) if both attribute types are enumeration classes, mark them as correct. Otherwise, it is partially correct, and (3) if both types are primitive, mark them as correct. Otherwise, they are marked as partially correct. Attribute matches with correct type checking gain scores of one. Otherwise, their scores are set to 0.5. A similar strategy is also used for *Stage 2.2*, with the maximum score being 0.5 because attributes are in different classes.

Stage 2.2: Attribute Matching Between Any Classes. Some attributes can be misplaced in incorrect classes but they can still make sense to the overall modeling effort. 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. We want 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. A weighted average is used in combining pair-wise attribute-to-attribute and class-to-class similarity into one value. Once the combined cosine similarity is obtained, they are passed to the `match_attributes` function for matching.

Stage 2.3: Reference Attribute to Candidate Class Matching. This stage is intended to identify any match between a reference attribute and a candidate class. Leftover reference attributes and leftover candidate classes are collected into two separate lists. Two sets of similarity scores are created, i.e., 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 between the reference attributes and the candidate classes are calculated. Reference attributes and candidate classes are embedded with a word embedding approach and our algorithm computes 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 match, since this is an anti-pattern, the score is set to 0.5 only.

Stage 2.4: Reference Class to Candidate Attribute Matching. In *Stage 2.4*, our algorithm finds a reference class matching a candidate attribute. It is similar to *Stage 2.3* whereas the class-to-attribute similarity is calculated between leftover reference classes and leftover attributes. The other steps are identical to *Stage 2.3*.

3.3 Stage 3: Relation Matching

In *Stage 3*, our algorithm matches relations based on class matches. Relation matching is rule-based, which is different from class or attribute matching. The algorithm iterates through all combinations of relations from the candidate model and the reference model. For each relation in the reference model, a list called `matchings` is initialized. Then, for each relation in the candidate model, if both relations are unmatched, a potential match is examined using the `compare_edges` function. The matching is appended to the list of `matchings` if it is a partial or exact match. After comparing all possible combinations, the best match among the matches is found.

Function `compare_edges`. This function takes two relations and tries to find an *exact match* (worth 1 point), a *partial match* (0.5 points), or *no match* (0 points). There are three kinds of relations defined in our language: `inherit`, `contain`, and `associate`. An `inherit` relation only consists of three elements: child class, inherit, and parent class. Both `associate` and `contain` relations consist of five elements: multiplicity 1, class 1, relation type, multiplicity 2, and class 2. Therefore, the input relations are classified based on the number of elements. If both relations contain three elements, they enter *Stage 3.1* Inheritance matching. In this stage, we check whether the second element is `inherit`. Meanwhile, we check if the first elements in both relations have already been matched in previous stages. The third elements are also checked to

confirm whether they have been matched. If all three conditions are satisfied, an exact match between these two relations is established. Meanwhile, if both relations 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. If the above exact match fails, the function ignores multiplicities and types but only checks whether classes are matched. Consider the reference relation * ControlCommand associate 1 ActuatorDevice and candidate relation * ControlCommand associate 1 Actuator as an example. `ControlCommand` is matched with `ControlCommand`. `ActuatorDevice` is matched with `Actuator`. All the other elements are also matched with their counterparts. This is an exact match. If there were mismatches in multiplicities, it would be a partial match.

3.4 Stage 4: Statistical Results

In *Stage 4*, our algorithm calculates precision, recall, and F_1-scores for the assessed domain model. Following our previous work [8], we use the same approach for calculating the statistics. For example, let C be the set of all classes in the candidate model of size $m = |C|$, the precision of classes in the candidate model can be expressed as

$$Precision_C = \frac{\sum_{i=0}^{m} \mathcal{S}_i}{m}, \tag{5}$$

where S_i is the matching score $\in [0, 0.5, 1]$ for the i-th class.

Recall measures the degree of the reference model covered by the candidate model. Let C be the set of classes in the *reference model* of size $n = |C|$, the recall of classes can be expressed as

$$Recall_C = \frac{\sum_{i=0}^{n} \mathcal{S}_i}{n}. \tag{6}$$

Finally, we use the classical F_1-score definition:

$$F_1^C = \frac{2 \times Precision_C \times Recall_C}{Precision_C + Recall_C}. \tag{7}$$

Metrics for attributes or relations can be computed by substituting C with the set of attributes or relations.

3.5 Grading

A weighted average of F_1-scores is used to produce the final grade of the domain model:

$$grade = \frac{w_c}{w_c + w_a + w_r} F_1^C + \frac{w_a}{w_c + w_a + w_r} F_1^{\mathcal{A}} + \frac{w_r}{w_c + w_a + w_r} F_1^{\mathcal{R}} \tag{8}$$

where w_c, w_a, and w_r are weights for classes, attributes, and relations, respectively. F_1^C, $F_1^{\mathcal{A}}$, and $F_1^{\mathcal{R}}$ are the F_1-scores for classes, attributes, and relations, respectively. Motivated by our real-world grading practice, the final grade is calculated with weights $w_c = 4$, $w_a = 1$, and $w_r = 1$.

4 EVALUATION

4.1 Evaluation of Generated Matches

This section explains how to evaluate our algorithm by comparing matches from our algorithm with matches from a human grader.

Comparison of Matches. For an assessed domain model, there is a collection of matches produced by the algorithm. There is another collection of matches produced by a human grader. Let us define two lists of matches as M_A (algorithm matches) and M_H (human matches). Each individual match: m_i is reformulated in the form of `(element_name, counterpart, matching_score)`. The first element in a match is always an element from the reference model or `None` for consistency. For conciseness, we only present the first two elements in the later description.

Labeling Strategy. Each match pair is classified as: true positive `TP`, false positive `FP`, true negative `TN`, and false negative `FN`. We focus on the first two elements in the match in *Match Comparison*, and consider the matching score in *Score Assignment*.

Match Comparison. Let $m = (a, b)$ be a match where a represents a model element from the reference model, and b represents a model element from the candidate model (where one of them may be $None$). Let m_a denote a match from M_A and m_h be a match from M_H. Then we demonstrate all the situations of m_h and m_a pairs and their evaluation labels as follows:

(1) (a, b) vs. (a, b). The human grader matches element a with element b, and our algorithm also matches element a with element b. This is a true positive (`TP`) with a score of 1.
(2) (a, b) vs. $(a, None)$. Our algorithm matches element a with nothing (i.e., $None$). However, a is supposed to be matched with something (i.e., b). Therefore, we consider this as a false negative (`FN`) with a score of 0.
(3) (a, b) vs. (a, c). Our algorithm matches element a with c. However, a is supposed to be matched with another element b. Thus, this is a false positive (`FP`) with a score of 0.
(4) $(a, None)$ vs. (a, b). Our algorithm matches an unnecessary element to a. This is a false positive (`FP`) with a score of 0.
(5) $(a, None)$ vs. $(a, None)$. Both the human grader and our algorithm also match element a with nothing. Therefore, we consider this as a true negative (`TN`) with a score of 1.
(6) $(None, a)$ vs. (b, a). Same as (4) with an `FP` and a score of 0.
(7) $(None, a)$ vs. $(None, a)$. Same as (5) with a `TN` and a score of 1.
(8) (b, a) vs. $(None, a)$. Same as (2) with an `FN` and a score of 0.
(9) (c, a) vs. (b, a). Same as (3) with an `FP` and a score of 0.

Score Assignment. For match pairs labeled as `TP` or `TN`, we further examine whether the third element `matching_score` is consistent in both matches. If they are consistent, a score of one is assigned to the match pair. If not, a score of 0.5 is assigned.

Evaluation Workflow. With the labeling strategy, we proceed to Step 1: *Find TN and TP*. This step is finding the intersection of M_A and M_H. For any pair of matches $(m_a, m_h) \in (M_A, M_H)$, m_a and m_h are considered if the first two elements of m_a are identical to the first two elements of m_h, i.e., $m_{a_1} = m_{h_1} \wedge m_{a_2} = m_{h_2}$. After finding a pair of identical matches, we need 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`.

Next, we proceed to Step 2: *Find FN and FP*. At first, we focus on Step 2.1 *Leftover Human Matches*, which is to find FN or FP in the leftover M_H. Step 2.1 can be further divided into two small steps. (1) For each leftover human match m_h with elements (a, b), we can try to find an algorithm match m_a with elements (a, x) where $x \neq b$. This is finding an algorithm match based on the first element. (2) If we cannot find such an m_{aj}, then we shift our focus to the second element and try to find an algorithm match m_a with elements (x, b) where $x \neq a$. For each match, we classify it into `FP` or `FN`.

There may still exist some leftover algorithm matches. Therefore, In Step 2.2: *Leftover Algorithm Matches*. We apply the same strategy as for Step 2.1 as described in the previous paragraph. Eventually, we

collect a list of match pairs with labels and calculate the precision, recall, and F_1-score with the labeled match pairs.

4.2 Evaluation of Generated Statistics

We evaluate our algorithm by comparing the generated statistics with those from human graders. Our algorithm provides statistics including precision, recall, and F_1-scores for each type of model element, along with final grades. It should generate statistics that are as close as possible to the statistics produced by human graders. To evaluate the performance of our algorithm in generating meaningful statistics, we apply the mean absolute error (MAE) metric. An MAE close to 0 indicates that our algorithm is closely aligning the results of human graders. Pearson correlation is another metric. A correlation value approaching 1 indicates a strong alignment.

5 EXPERIMENTS

This section aims to evaluate our algorithm's performance in assessing domain models and identify areas of the matching and grading task where our algorithm may encounter challenges. We aim to investigate the following research questions (RQ): (1) **what is the performance of our algorithm in matching a candidate domain model to a reference model regarding classes, attributes, and relations?** and (2) **to what extent does the algorithm-generated grade compare with those produced by human grading or other automated approaches?**

5.1 Experimental Settings

We have chosen a modeling problem on the smart home domain that was an assignment in an undergraduate-level modeling course. The problem description and reference domain model are in the supplementary material[1]. 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. There are 5 enumeration classes, 15 regular classes, 3 abstract classes, 13 enumeration literals, 13 attributes, and 32 relations in the reference domain model. We randomly select 20 student solutions. To establish the ground truth, one author of this paper manually matches candidate models against the reference model. To ensure fairness of the evaluation, the manual match is done before having seen matches from the algorithm.

Our algorithm relies on some external libraries for embeddings and graph comparison. We select `sgram_mde` (skip-gram for MDE) from the WordE4MDE library as the word embedding method. In terms of sentence embedding, we select one OpenAI embedding model, `text-embedding-ada-002`, via the API provided by OpenAI[2]. Calculating graph edit distance is also essential to our algorithm. We use the Python library NetworkX [11] whose GED algorithm is developed based on the work by Abu-Aisheh et al. [2].

5.2 RQ1: Matching Performance

RQ1 aims to evaluate how our algorithm performs in matching model elements from the candidate model to elements from the reference model. We carry out a more fine-grained analysis by highlighting with which modeling aspects the algorithm struggles.

[1]https://github.com/ChenKua/Domain_Model_Elevation
[2]https://platform.openai.com/docs/guides/embeddings/embedding-models

Table 2: Average performance scores for matching

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
Relation	Precision	0.8556 ± 0.1492
	Recall	0.7628 ± 0.1847
	F_1-score	0.7958 ± 0.1505

Particularly, we compare the performances of the algorithm when matching classes, attributes, and relations.

The performance scores in Table 2 compare the matching results generated by our algorithm with human-generated matches of 20 student submissions. Our algorithm excels in class matching (F_1-score of 0.8239), surpassing those for attribute matching (0.7456) and relation matching (0.7958). Notably, the F_1-score for attribute matching is the lowest among the three element types. The precision of relation matching is the highest among the three, while the recall of class matching highly surpasses those of the other two. Based on the trend of F_1-scores, it can be inferred that our algorithm excels in matching classes compared to matching relations and it performs better in matching relations than in matching attributes. Additionally, the recall for matching classes and attributes is higher than their respective precision, while the phenomenon is reversed for matching relations. Class matching and attribute matching share a similar logic, whereas relation matching is rule-based with dependency on class matching. Relation matching relies on class matching, which might be why relation matching has higher precision than recall.

Answer to RQ1. While our algorithm demonstrates impressive capability in matching model elements, there is still room for performance improvement. Our algorithm achieves F_1-scores of 0.82 for class matching, 0.75 for attribute matching, and 0.80 for relation matching. Moreover, our algorithm struggles the most with matching attributes (compared to classes and attributes).

5.3 RQ2: Grading Performance

Our algorithm produces a set of statistics as numerical assessments for each assessed domain model. RQ2 seeks to explore whether these statistics can serve as meaningful grades for evaluated domain models. We aim to investigate the extent to which our algorithm is capable of generating statistics that reasonably approximate the ground truth values for grades. There are two types of comparison: *internal comparison* and *external comparison*.

RQ2.1: Internal Comparison. In RQ2.1, we conduct an *internal comparison* on algorithm-generated precision, recall, F_1-scores, and grades. Table 3 shows the mean absolute error (MAE) between precision, recall, and F_1-scores generated from our 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%. Our algorithm combines three F_1-scores through weighted averaging to derive the final grade. Employing identical weights as those utilized in our algorithm, we compute the weighted average of F_1-scores from human graders, thereby establishing them as the ground-truth

Table 3: The Mean Absolute Error (MAE) and Pearson Correlation between algorithm-generated data and human grading

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
Relation	Precision	0.04110	0.8272
	Recall	0.02891	0.8306
	F_1-score	0.03397	0.8006
	Grade	0.03096	0.8714

F_1-based grades. Consequently, we calculate the MAE between the algorithm-generated grades and the ground truth F_1-based grades, as shown in the final row of Table 3. This computed MAE stands at 0.03096, indicating a small deviation. The Pearson correlation between metrics derived from our algorithm and those from human graders is computed and exhibited in the last column of Table 3. All correlation coefficients surpass 0.8, indicating a robust correlation between metrics generated by our algorithm and those by humans.

Answer to RQ2.1. Our algorithm excels in accurately evaluating domain models, demonstrating precision, recall, F_1-scores, and grades that closely align with the human grading. Across all metrics, the Mean Absolute Error (MAE) remains consistently below 0.05 (i.e., within one letter grade range), accompanied by strong correlations exceeding 0.8.

RQ2.2: External Comparison. RQ2.2 evaluates how the grade produced by our algorithm approximates the grades from an existing external benchmark from Singh et al. [20]. Modeling grades in this benchmark are produced by domain modeling experts. We compare our algorithm-generated grades with the same set of domain model grading results in the benchmark. 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. We also apply GPT-4-turbo to grade the same set of student submissions, which results in an MAE of 0.0674. Furthermore, we test another domain model assessment approach in TouchCore [5], which fails to work on 20% of submissions and yields an MAE of 0.2524 for the other 80% (with weights of model elements adjusted to TouchCore's settings). Our algorithm improves MAE significantly compared to the other two approaches. 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 RQ2.2. Our approach clearly outperforms the baseline state-of-the-art auto-grading approach. It 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 our data and the benchmark is 0.0456, indicating a difference of about 5 points out of 100.

5.4 Discussion

RQ1. The experiments show that our algorithm is still not able to match the model elements as well as a human grader. The results obtained are promising but there is still room for improvement. The results in RQ1 reveal that our algorithm struggles to match attributes correctly. Meanwhile, class matching and attribute matching, both suffer from low precision and high recall. This suggests a higher percentage of false positives compared to false negatives in the matches. One future direction can be reducing the false positives by defining more rule-based or embedding-based mechanisms to identify None matches, i.e., `(a, None)`.

RQ2. By addressing RQ2, we aim to provide insight into how grades produced by our algorithm correlate with human grading. We have observed that our algorithm's outputs exhibit a high degree of correlation with human grading values. Our algorithm also demonstrates its effectiveness by generating meaningful precision, recall, F_1-scores, and grades that closely align with the ground truth values in both internal and external benchmarks, indicating its capability to assess domain models accurately. While our algorithm still cannot replace human graders, it produces explainable matches of model elements, which may be a useful starting point for graders' evaluations and a self-assessment tool for students in practice.

5.5 Threats to Validity

Internal Validity. The authors only construct a small matching data set, which may introduce bias and insufficient statistical power. We mitigate this bias by also comparing our algorithm outputs against an external human-grading benchmark [20]. The selection of scores for model elements and weights for F_1-scores influences the final grades. We choose the scores following our previous work and select weights widely used to evaluate university-level assignments. The algorithm's outputs are evaluated using scores of {0, 0.5, 1}, which is commonly used in university grading settings. There are numerous ways to represent a domain model. Following our previous work, we apply the same model representation.

External Validity. There are benchmarks [20] for grading in the automated domain model assessment research. However, to the best of our knowledge, there are no benchmark data sets for model element matching. Thus, one author of this paper curates such a data set manually, leading to a higher risk of getting false positives in the statistical tests. The scalability of the algorithm is yet to be evaluated, especially considering the lower bound of the used GED algorithm is $O(n^3)$. There may be risks with changes in the OpenAI embedding models. To ensure reproducibility, we have saved the embedding results of the model elements used in our experiments. All experiments were conducted on English datasets, which may affect the generalizability of our algorithm to other language settings.

Construct Validity. Our paper adapts metrics widely used for classification and numerical comparison [4, 8, 17, 20, 23].

6 RELATED WORK

Many approaches have been investigated to assess the candidate model in comparison to the reference model. In general, they can be divided into four categories.

Rule-based. Bien et al. [5] 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. Singh et al. [20] 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) present in a submission. Model comparison is a critical process in MDE, allowing developers to identify differences between various versions of models. EMFCompare is a tool from the Eclipse Modeling Framework (EMF) that facilitates model comparison and merge tasks by providing a comprehensive set of functionalities for detecting and resolving differences [1]. EMF Diff/Merge is another tool providing a lightweight, configurable approach for model comparison and merging with GUI components [10]. SiDiff is also a model comparison tool which also utilizes the similarity of model elements for comparison [21].

Graph Matching. Tselonis et al. [22] 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. One of the matching algorithms investigated by the authors is graph isomorphism, which involves determining whether a diagram is either isomorphic to the reference answer or contains an isomorphic subgraph of it. Similarly, Ludovic et al. [3] also apply a graph-matching approach for assessing class diagrams. Their 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. Boubekeur et al. [6] 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.

Our approach combines rule-based, graph matching, and machine learning to produce a score for candidate models based on different metrics such as precision, recall, and F_1-score. It is more flexible and considers semantics by using text embeddings compared to rule-based approaches. No training is required for text embeddings. It is also more scalable compared to graph matching and taking attributes, and class names into consideration.

7 CONCLUSION

This paper introduces a novel algorithm for fully automated domain model assessment utilizing text embeddings and graph comparison techniques. The algorithm automatically matches model elements between a candidate and reference model, subsequently generating a grade for the candidate model without human intervention. Our experiments indicate that although the proposed algorithm exhibits impressive performance in matching model elements and scoring domain models, there remains potential for enhancement. In future work, we aim to leverage the explainable matches to generate human-readable feedback that is both meaningful and immediate, thereby enhancing the learning experience for students.

REFERENCES

[1] 2024. EMF Compare. https://eclipse.dev/emf/compare/. Accessed: 2024-06-30.

[2] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. 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* (Lisbon, Portugal) *(ICPRAM 2015)*. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 271–278.

[3] Ludovic Auxepaules, Dominique Py, and Thierry Lemeunier. 2008. 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)*. IEEE, USA, 26–30.

[4] Viv Bewick, Liz Cheek, and Jonathan Ball. 2003. Statistics review 7: Correlation and regression. *Critical care* 7 (2003), 1–9.

[5] Weiyi Bian, Omar Alam, and Jörg Kienzle. 2021. Automated grading of class diagrams. In *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems* (Munich, Germany) *(MODELS '19)*. IEEE Press, Munich, Germany, 700–709.

[6] Younes Boubekeur, Gunter Mussbacher, and Shane McIntosh. 2020. 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* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 10 pages.

[7] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. 2023. Towards Using Few-Shot Prompt Learning for Automating Model Completion. In *Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '23)*. IEEE Press, Melbourne, Australia, 7–12.

[8] Kua Chen, Yujing Yang, Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Dániel Varró. 2023. 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)*. IEEE, Västerås, Sweden, 162–172.

[9] Xiaoyang Chen, Hongwei Huo, Jun Huan, and Jeffrey Scott Vitter. 2019. An efficient algorithm for graph edit distance computation. *Knowledge-Based Systems* 163 (2019), 762–775.

[10] Eclipse Foundation. 2024. Eclipse EMF Diff/Merge. https://projects.eclipse.org/projects/modeling.emf.diffmerge. Accessed: 2024-06-30.

[11] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[12] Jiawei Han, Micheline Kamber, and Jian Pei. 2012. 2 - Getting to Know Your Data. In *Data Mining (Third Edition)* (third edition ed.), Jiawei Han, Micheline Kamber, and Jian Pei (Eds.). Morgan Kaufmann, Boston, 39–82.

[13] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, Vol. 1. Association for Computational Linguistics, Minneapolis, MN, USA, 2.

[14] José Antonio Hernández López, Carlos Durá, and Jesús Sánchez Cuadrado. 2023. Word Embeddings for Model-Driven Engineering. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, IEEE, Västerås, Sweden, 151–161.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013), 1–9.

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[17] Rijul Saini, Gunter Mussbacher, Jin L. C. Guo, and Jörg Kienzle. 2022. Machine Learning-Based Incremental Learning in Interactive Domain Modelling. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada). ACM, New York, NY, USA, 176–186.

[18] Robert Sedgewick and Michael Schidlowsky. 2003. *Algorithms in Java, Part 5: Graph Algorithms* (3 ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[19] Oszkár Semeráth, Rebeka Farkas, Gábor Bergmann, and Dániel Varró. 2020. Diversity of graph models and graph generators in mutation testing. *International Journal on Software Tools for Technology Transfer* 22 (2020), 57–78.

[20] Prabhsimran Singh, Younes Boubekeur, and Gunter Mussbacher. 2022. Detecting mistakes in a domain model. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 257–266.

[21] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. 2007. Difference computation of large models. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 295–304.

[22] Christos Tselonis, John Sargeant, and Mary McGee Wood. 2005. Diagram Matching for Human-Computer Collaborative Assessment. In *Proceedings of the 9th International Computer Assisted Assessment Conference*. Loughborough University, UK, 1–15.

[23] Cort J Willmott and Kenji Matsuura. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate research* 30, 1 (2005), 79–82.