Université de Montréal

Extraction of UML Class Diagrams from Natural Language Specifications

par

Song Yang

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Discipline

November 15, 2022

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Extraction of UML Class Diagrams from Natural Language Specifications

présenté par

Song Yang

a été évalué par un jury composé des personnes suivantes :

Stefan Monnier
(président-rapporteur)

Houari Sahraoui
(directeur de recherche)

Michalis Famelis
(membre du jury)

Résumé

Dans l'ingénierie dirigée par modèle, les diagrammes de classes UML servent à la planification et à la communication entre les différents acteurs d'un projet logiciel. Dans ce mémoire, nous proposons une méthode automatique pour l'extraction des diagrammes de classes UML à partir de spécifications en langues naturelles. Pour développer notre méthode, nous créons un dépôt de diagrammes de classes UML et de leurs spécifications en anglais fournies par des bénévoles. Notre processus d'extraction se fait en plusieurs étapes: la segmentation des spécifications en phrases, la classification de ces phrases, la génération des fragments de diagrammes de classes UML à partir de chaque phrase, et la composition de ces fragments en un diagramme de classes UML. Nous avons validé notre approche d'extraction en utilisant le dépôt de paires diagramme-spécification. Même si les résultats obtenus montrent une précision et un rappel bas, notre travail a permis d'identifier les éléments qui peuvent être améliorés pour une meilleure extraction.

Mots clés: Génie logiciel, UML, Ingénierie dirigée par modèle, Extraction d'information, Informatique

Abstract

In model-driven engineering, UML class diagrams serve as a way to plan and communicate between developers. In this thesis, we propose an automated approach for the extraction of UML class diagrams from natural language software specifications. To develop our approach, we create a dataset of UML class diagrams and their English specifications with the help of volunteers. Our approach is a pipeline of steps consisting of the segmentation of the input into sentences, the classification of the sentences, the generation of UML class diagram fragments from sentences, and the composition of these fragments into one UML class diagram. We develop a quantitative testing framework specific to UML class diagram extraction. Our approach yields low precision and recall but serves as a benchmark for future research.

Keywords: Software engineering, Model-driven engineering, Information extraction, Natural language processing, Machine learning, UML

Contents

List of tables

List of figures

List of abbreviations

MDE	Model-driven engineering, a style of software development based on modeling
UML	Unified Modeling Language, a standard adopted by the Object Management Group for visual diagrams
OOP	Object-oriented programming, a programming language paradigm that treats classes and objects as first-class citizens
NLP	Natural language processing, a field of computer science dealing with human language and linguistics
POS	Part of speech, a linguistic feature of words, such as nouns, verbs, adjectives, conjunction, etc.
AI	Artificial intelligence, a computational agent

Chapter 1

Introduction

1.1. Context

Software is more and more present in our daily lives. Not only is there more software, but the software itself is also becoming increasingly complex. For example, the Apollo Program that sent the first human to the Moon had 600k lines of code. In comparison, Microsoft had 200 million lines of code for their main services by 2010 [**36**]. Hence, the process of making software is becoming complex as well.

When making software, developers need to take care of low-level concerns, such as libraries and programming languages, as well as high-level concerns, such as functionality and user experience. Thinking about all of this at once for real-world applications is unfeasible. As such, model-driven engineering (MDE) has been introduced as an efficient approach for reducing the complexity of software development by increasing the level of abstraction [**19**].

MDE makes use of a model, which is an abstraction of a running system, to understand, communicate and analyze software-intensive systems [**35**]. One way of describing a model is through the Unified Modeling Language (UML), which is a visual language used to specify, construct and document software systems. A UML class diagram is the most used UML diagram. It shows a static view of the system, consisting of classes, their interrelationships (including generalization/specialization, association, aggregation, and composition), operations, and attributes of the classes [**32**].

The creation of a UML class diagram requires a software specification, which in turn is obtained when the developer interviews their customer. This thesis deals with the difficulty of creating a UML class diagram from a natural language specification.

1.2. Problem

Modeling a UML class diagram from a specification is hard, especially when developers engage in a domain that they have little or no knowledge of. Since modeling requires skill

and experience, several approaches have been proposed to automate the extraction of models from natural language. One recent approach is DoMoBOT [**29**]. However, this tool is not fully automated because it requires the user to provide interactive feedback. A survey of existing publications concluded that most proposed tools are not fully automated and require consistent user intervention. And if they are automated, most tools require the specification to be written in a specific form or use a restricted subset of English [**2**]. For example, [**1, 20**] are automated but they require the English specification to be written in a certain way.

This thesis addresses the shortcomings by proposing a tool that is fully automated and that accepts unrestricted natural language input.

1.3. Contributions

This thesis contributes a tool to make the software development process better. The tool automatically turns a specification written in English into a UML class diagram. More specifically, the contributions are:

- An automated pipeline of natural language processing (NLP) and machine learning techniques
- A set of English linguistic grammar patterns to detect common expressions in a specification
- An algorithm to merge simple UML class diagrams into one larger UML class diagram
- A binary classifier to detect what part of a UML class diagram a sentence is describing
- A dataset of UML-English pairs, where each pair consists of a UML class diagram fragment and an English specification describing the fragment
- An executable that performs the pipeline of translating English into UML class diagrams

1.4. Thesis structure

Chapter 2 gives a background in model-driven engineering and natural language processing (NLP) and surveys the related work in feature extraction from natural language specifications. Chapter 3 details the approach used to build the NLP pipeline, including the data collection of the novel dataset, preprocessing, machine learning, and NLP grammar techniques. Chapter 4 discusses the performance of the translation. Chapter 5 concludes the thesis and provides avenues for future work.

Chapter 2

Background and Related Work

Introduction

In this chapter, we provide the necessary background to understand the contributions of this thesis and summarize the related work in model-driven engineering and natural language processing. Finally, we show how our system to extract UML class diagrams from English differs from the related work.

2.1. Definitions and examples

2.1.1. Software specification

Software requirements specification denotes a precise description of the system objects, a set of methods to manipulate them, and a statement on their collective behavior for the duration of their existence in the system to be developed [**4**]. A specification can be obtained by speaking to the customer.

> A contact has a name. The phone number of a land line is encoded as a String. A cell phone number is mapped to a single contact. The number of a cell phone number is encoded as a String. A Contact may have any number of land line numbers. A land line number is mapped to exactly one contact.

Fig. 2.1. An example of a short software specification

Figure 2.1 shows an example specification for a software system requested by a customer interested in managing personal contact information. Software specification is not limited to text. There can be images and recordings. But in this thesis, we are only interested in textual descriptions.

A specification is considered "good" when it describes what is needed rather than how to implement it [**5**]. There is a possibility that a customer doesn't know what they want or how to express ideas properly. As such, a specification written directly by the customer might require further clarification. In this thesis, we make use of good specifications.

2.1.2. Model (Software engineering)

In software engineering, a **model** is an abstraction of a running system, to understand, communicate and analyze software-intensive systems [**35**]. This can range from whiteboard drawings to more formal representations that can generate code. The Object Management Group has defined many kinds of models, most notably the Unified Modeling Language (UML). If a model describes models, then it is a **meta-model** [**15**].

2.1.3. Model-driven engineering (MDE)

MDE is a field of software engineering. It refers to all methodologies that are model-centric and which encourage efficient use of models during all stages of software development. MDE encourages the use of abstraction to improve understanding, fit the needs of the domain, and automate repetitive tasks. A case study in the industry concluded that MDE, while useful in some regards, is not widely adopted due to poor ease of use and a lack of mature tools [**19**].

2.1.4. Unified Modeling Language (UML)

UML is a family of graphical notations, backed by a single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented style [**10**]. There are many kinds of UML diagrams. The most used one is the class diagram [**32**].

UML can be used as a sketch to communicate between developers [**10**]. It can also be used as a blueprint to plan out the code, check for consistencies and potentially generate code from it [**7**], such as Java [**22**]. Furthermore, UML can be used for reverse-engineering [**12**].

2.1.5. UML class diagram

A **UML class diagram** is a kind of UML diagram that shows a static view of a system, consisting of classes, their interrelationships (including generalization/specialization, association, aggregation, and composition), operations, and attributes of the classes. The UML class diagram can be presented in three perspectives, which are the conceptual perspective, the specification/design perspective, and the implementation perspective [**32**].

The perspective of interest in this thesis is the specification/design perspective. In this perspective, the diagram is interpreted as a description of software abstraction, without a particular implementation in mind [**32**].

Fig. 2.2. Example of a UML class diagram

Figure 2.2 is an example of a UML class diagram. In this diagram, there are two classes, represented by boxes, and one unidirectional association, represented by the arrow. In general, there is a nonzero number of classes and any number of associations. The "variableName: type" entry inside classes represents an attribute of name "variableName" and type "type". In a UML class diagram, there is an arbitrary number of attributes, including none at all, and the presence of attribute types is optional.

"associationA" over the arrow is the name of a unidirectional association. The "0..*" (pronounced "zero to many") under the arrow is a **multiplicity** and means that instances of ClassA can be associated with any number of instances of ClassB, including none at all. In Figure 2.2, the association reads as "Class A is associated to Class B under the 'associationA' relationship, where for each Class A there are zero-to-many Class B."

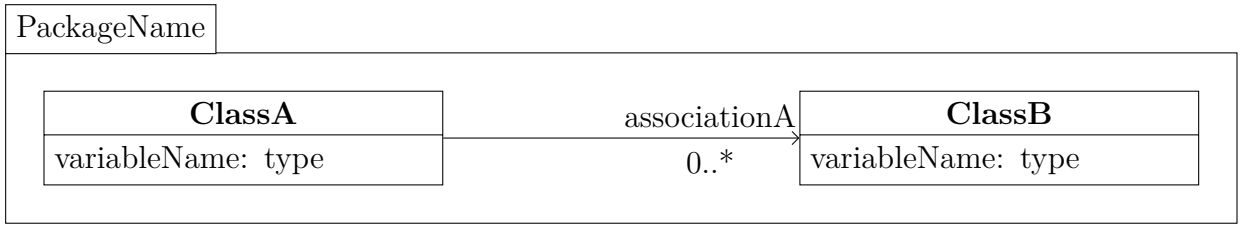


Fig. 2.3. Example of a UML class diagram with a package

Figure 2.3 shows the possibility of having a package inside a UML class diagram. A UML package is used to denote some groupings within a diagram, for example to help with clarity.

A UML class diagram can have other kinds of associations and other entities than classes. For example, inheritance, which allows new object definitions to be based upon existing ones [**33**], can be drawn between two classes. In this thesis, we will use a simplified version of the UML class diagram that makes use of only classes, unidirectional associations, and packages, such as Figure 2.3.

2.1.6. Natural language processing (NLP)

NLP is a field of computer science. It is a collection of computational techniques for automatic analysis and representation of human languages, motivated by theory [**8**]. An example use of NLP is Google Translate.

There are two broad camps in NLP. The first camp is older and makes use of statistical methods, such as the N-gram based on a multi-order Markov model [**21**]. The second camp is newer and makes use of neural networks from machine learning, such as word vectors [**30**]. In this thesis, we use mostly statistical methods due to the lack of sufficient data for neural methods. We have tried fine-tuning an existing neural model called BERT [**9**] on our dataset, and the results were very poor.

2.1.7. Part-of-speech (POS) tagging

Part-of-speech tagging tries to tag (or label) each word in a sentence with the correct part of speech [**16**]. A POS is the words of a language that can be collected into classes of formal equivalents [**6**], such as nouns, verbs, adjectives, etc.

Figure 2.4 shows an example of a tagging result. The tags are shown in color under the words of the sentence being parsed.

2.1.8. Dependency parsing

Dependency parsing is a form of syntactic parsing of natural language based on the theoretical tradition of dependency grammar. A dependency is between a syntactically subordinate word, called the *dependent*, and another word on which it depends called the *head* [**24**].

Figure 2.4 shows the example of a parse result for the sentence "The tree has grown taller". In the figure, the capitalized words under the sentence are the POS tags, whose complete list can be found here[1]. In addition, arrows represent the dependencies by connecting words to each other. Under each arrow is the abbreviation of a grammatical function. For example, "nsubj" means the word "tree" plays the role of a nominal subject for "grown". The complete list of abbreviations can be found here[2].

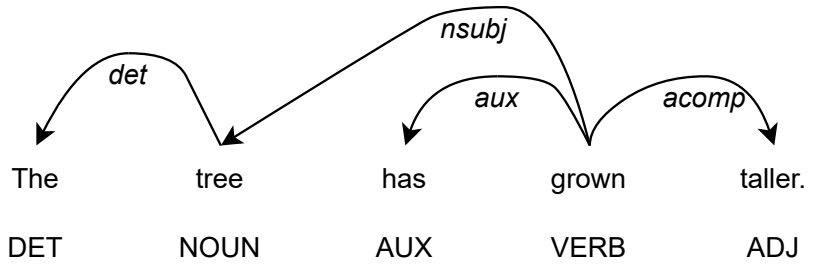


Fig. 2.4. Example of a sentence's parse tree containing POS and dependency information

[1] https://machinelearningknowledge.ai/tutorial-on-spacy-part-of-speech-pos-tagging/
[2] https://github.com/clir/clearnlp-guidelines/blob/master/md/specifications/dependency_labels.md

2.1.9. Lemmatization

Lemmatization is the process of finding the normalized form of a word. It is the same as looking for a transformation to apply to a word to get its normalized form [**27**].

In this thesis, we perform lemmatization on nouns and verbs in an English sentence. Nouns are changed to their singular form, while verbs are changed to their non-conjugated (indicative) form. For example, "cities" is changed to "city" and "bought" is changed to "buy".

2.1.10. Vectorization (NLP)

Vectorization is the process of storing text in a specified vector space [**14**]. By doing this, we open more path ways to manipulate the data. Two broad categories of vectorization exist.

- Direct representations: Each vector component traces back to a specific item in a vocabulary. (ex. bag-of-words)
- Indirect representations: Each vector component does not trace back to any specific item in a vocabulary. (ex. word vectors)

Direct representations include the **bag-of-words** representation. A bag of words is a vector whose components represent the words in a vocabulary. A sentence or the entire document can be turned into a bag of words. The value of the vector components depends on the calculation method.

The most straightforward calculation method is count vectorization. According to this method, the components of a bag of words are the frequencies of the words in the document being vectorized. The count vectorization method has a bias for very common words. To mitigate this bias, the *term frequency-inverse document frequency* (TF-IDF) method penalizes very common words in a vocabulary, such as "the".

Indirect representations include **word vectors**. The creation of word vectors requires the training of a statistical model on a large third-party dataset of text [**17**]. With a trained model, every word can be transformed into a vector of some dimension. Each component in this vector cannot be linked to any words in the vocabulary, unlike direct representations. Instead, the components of a word vector represent a semantic meaning. In theory, synonyms should have very similar word vectors.

In this thesis, we use a direct representation due to its simplicity and speed.

2.1.11. Coreference resolution

A **coreference** is words or phrases referring to a single unique entity (or union of entities) in an operating environment. **Coreference resolution** is the resolution of repeated references to an object in a document [**31**].

One example of coreference resolution is finding what pronouns ("he", "she", "they", "it") refer to in a document. Another example is grouping expressions that refer to the same entity in a document. For example, expressions like "the country" and "the hockey place" may refer to Canada in a document.

2.2. Related Work

In this section, we summarize the existing research on the extraction of UML class diagrams from natural language specification. Without automation, developers produced UML class diagrams from specifications manually. Several recurring patterns in software engineering can be identified in specifications. In [**23**], Nicola et al. describe common situations when talking to business analysts. In the rest of this section, we talk about the state-of-the-art of automatic and semi-automatic methods of extracting UML class diagrams.

2.2.1. Survey of relevant literature

In 2021, Abdelnabi et al. surveyed 24 published tools and methods for the extraction of UML class diagrams from natural language specifications. In the survey, most tools and methods require consistent user intervention. Most tools also required the specification to be given in a specific format, such as a more restricted vocabulary of English or a more rigid structure rather than free-flowing text. The authors concluded that no fully automated tool to generate complex UML class diagrams exists [**2**].

Degree of automation	# of papers
Semi-automatic	9
Automatic	15

Input	# of papers
Unrestricted English	1
Restricted English	19
Structured format	4

Table 2.1. Results of the 2021 survey [**2**]

2.2.2. Automatic approaches

Automatic approaches do not require extensive user intervention. Once input is given, the user only needs to wait for a result. The automatic approaches make use of more traditional NLP techniques, such as hand-written rules and grammar parsing. The authors of [**1, 20**] use several heuristics to analyze the natural language specification.

In [**1**], Abdelnabi et al. describe an extraction process based on NLP techniques. Their approach requires the user to keep a set of normalization rules in mind when writing a software specification. For example, one rule prohibits sentences from using conjunctions to connect objects after the verb. That means a sentence like "The customer has a name and a shipping address." is not allowed as input to their automated system. Once all rules

are satisfied, the automated part of their approach starts preprocessing the restricted input using NLP techniques that are similar to this thesis. Namely, the input is segmented by sentence, tokenized by word, and put through lemmatization (see Section 2.1.9). Then, the approach performs dependency parsing (see Section 2.1.8) based on a large number of grammar rules for identifying classes, attributes, methods, and relationships, respectively. There is no mention of how the parsing results are grouped into a UML class diagram. The approach has been evaluated subjectively in use cases and the results show it is "feasible" and "acceptable".

In [**20**], More et al. describe a tool named "RAPID", which is also an extraction process based on NLP techniques. Similar to [**1**], RAPID also requires its user to write the English specification in a restricted way, following a set of normalization rules. RAPID proceeds with similar preprocessing, such as lemmatization. There is no sentence segmentation because the user is expected to provide the input in a custom graphical environment. RAPID parses the input according to a smaller set of grammar rules for identifying classes, attributes, and relationships, respectively. In addition, RAPID makes use of synonyms via WordNet [**18**] to validate generalization relationships. There is also no mention of how the parse results are merged into a UML class diagram. RAPID's performance has not been evaluated.

2.2.3. Semi-automatic approaches

Semi-automatic approaches require extensive user intervention before a UML class diagram is generated. In [**29**], Saini et al. create an AI assistant called "DoMoBOT" that assists the user in creating UML class diagrams. The tool has a graphical user interface as shown in Figure 2.5.

Overall, DoMoBot's architecture is a complex blend of traditional NLP and machine learning techniques. After receiving an English specification, DoMoBOT preprocesses the text with sentence segmentation and coreference resolution (see Section 2.1.11). Then, DoMoBot extracts concepts from the text by identifying noun phrases and formatting them according to variable naming conventions with the help of lemmatization. The identification of relationships is done with dependency parsing from grammar patterns manually decided by Saini et al. Synonyms of previously extracted concepts are handled via similarity scores. This is their NLP rule-based part.

DoMoBOT also makes use of machine learning by using word vectors (see Section 2.1.10) to represent extracted concepts and sentences. Concepts are classified using a pre-trained neural network to predict if they are a UML class or a UML attribute and if they are an attribute, what types they have. The word vectors for sentences are grouped into matrices. Sentences are then classified using another pre-trained neural network to predict what kind

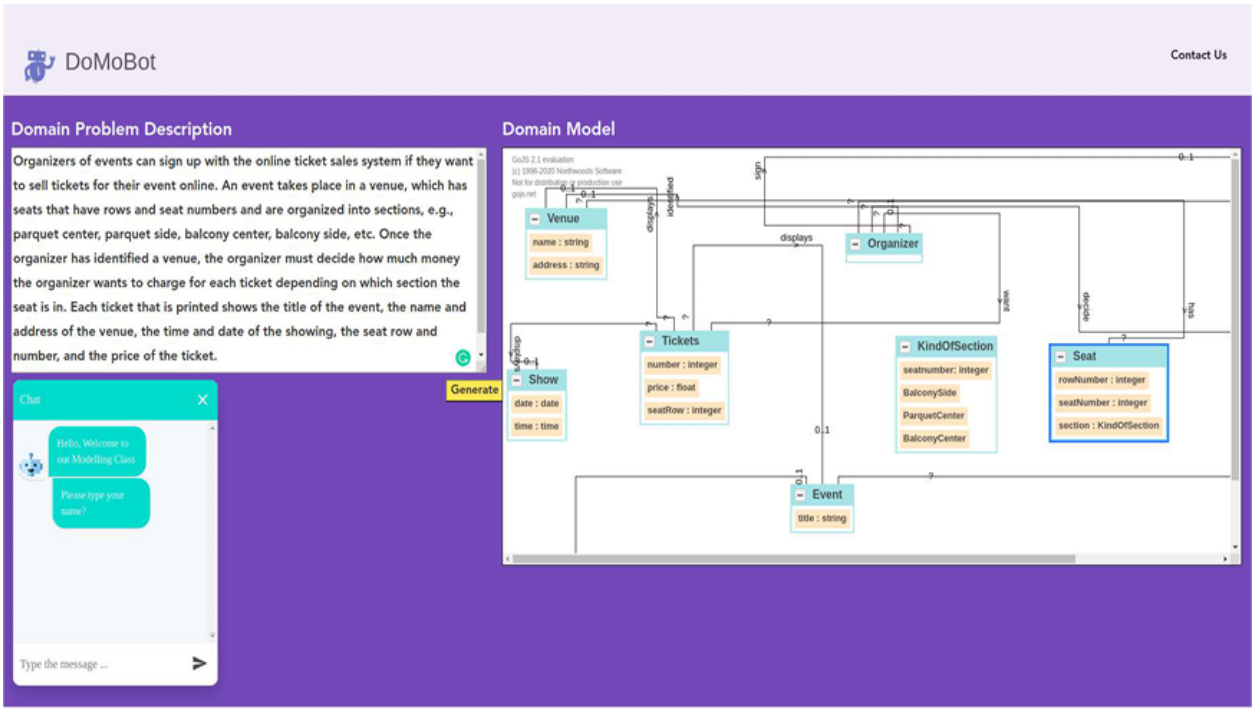


Fig. 2.5. DoMoBOT's graphical user interface [**29**]

of UML relationships they represent, such as association and composition, and to predict the multiplicities of the relationships.

The results from machine learning are grouped in clusters using an algorithm. If there are inconsistencies in the clustering, the result is presented with an "?" to the user who has to provide clarifications through the chat box. Responses received in the chat box go through DoMoBOT's system again. The process repeats until there are no more inconsistencies.

2.3. Synthesis

This thesis distinguishes itself from the related work in several ways.

2.3.1. Fully automated

While tools like DoMoBOT [**29**] can produce UML class diagrams, the user needs to provide constant input to guide the AI. The thesis proposes an automated method that requires minimal user intervention. Once the input is given, the user only has to wait for the output.

2.3.2. Unrestricted input

The automated approaches presented in [**1, 20**] have normalization rules, which require users to write specifications in a restricted English sentence structure. This thesis presents

an approach that accepts free-flowing text. We don't have any normalization rules that users must keep in mind.

2.3.3. Simpler heuristics and computational optimizations

Although [**1**] and [**20**] are automated and use grammar patterns and heuristics as well, our approach contains a fragmentation step. Instead of parsing all the input at once, the input is divided by sentence and processed separately, before being reassembled at the end. Our approach can handle very large texts and can reduce computation time by distributing fragments to concurrent processing units. In addition, the fragmentation simplifies the grammar patterns and heuristics even further.

2.3.4. Consistent testing metrics

The authors of the survey [**2**] did not provide a quantifiable metric for measuring the performance of UML class diagram extractors. Semi-automatic tools like DoMoBOT cannot have their performance quantified easily since it requires human intervention. Furthermore, the papers [**1, 20**] describing fully automated tools do not provide a guideline for quantifying performance. The authors of [**1**] performed a qualitative comparative evaluation with another tool. Hence, only qualitative assessments have been made.

As such, this thesis describes quantitative testing metrics for the evaluation of UML class diagrams obtained from English specifications. We provide a dataset that can be used for the evaluation of future UML class diagram extractors.

Chapter 3

From Specifications to UML Class Diagrams

Overview

The goal of this chapter is to design a method to translate English specifications to UML diagrams. To do this, we implement a tool pipeline that generates UML class diagrams from natural language specifications. The approach consists of two parts. The first part is the creation of a dataset. The second part is the NLP pipeline that performs the extraction of UML class diagrams. Figure 3.1 summarizes the process.

Our approach combines machine learning with pattern-based diagram generation. To perform machine learning, we start by creating a dataset of UML class diagrams and their

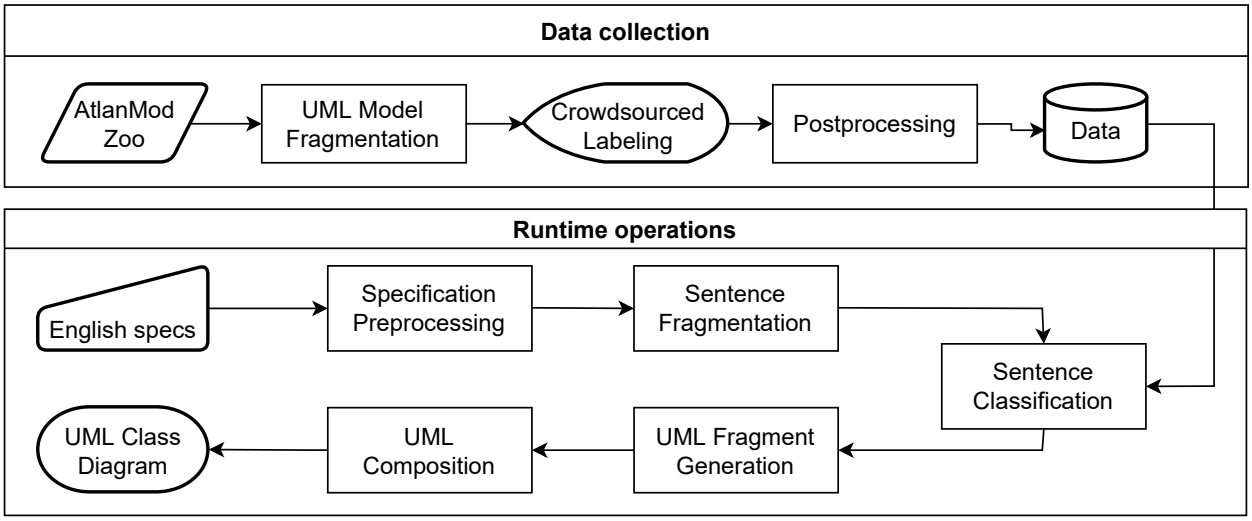


Fig. 3.1. Overview of the extraction process of UML class diagrams from natural language specification

corresponding specifications in natural language (top part of Figure 3.1). We selected pre-existing UML class diagrams from the AtlanMod Zoo repository [1]. The selected diagrams were decomposed into fragments and manually labeled by volunteer participants. After postprocessing, the labeled diagrams were stored in a repository.

The bottom part of Figure 3.1 consists of the diagram generation process, which takes place right after a user submits a software specification. The submitted natural language specification is then preprocessed and decomposed into sentences. Using a classifier built from the above-mentioned dataset, the sentences are labeled according to the nature of the UML construct they refer to, i.e., a class or a relation. According to this label, specific procedures of parsing and extraction are performed on the sentence to generate a UML fragment. In the end, all UML fragments are composed back together into one UML class diagram.

In the rest of this chapter, we detail all the activities of our approach.

3.1. Dataset Creation

We create a new dataset for both the operation and the evaluation of our approach. In particular, we use this dataset to learn a classifier for the Classification step in Figure 3.1.

To build the dataset, we start from an existing set of UML class diagrams from the AtlanMod Zoo. The AtlanMod Zoo has a repository of 305 high-quality UML class diagrams that model various domains. As defined in Section 2.1.5, a UML diagram contains classes and relationships. The size of the diagrams varies from a few classes to hundreds of classes. We fragment each diagram into UML fragments. A fragment is either a simple class (Figure 3.2) or a relationship (Figure 3.3) linking two attribute-less classes.

Table 3.1 shows the size of the dataset before and after labeling, as well as the quantity of class and relationship fragments. We only managed to label 7% of the dataset. The labeled portion is made of UML class diagrams with the least complexity, that is, with the least number of classes and relationships.

Dataset	UML models	UML fragments	Class fragments	Relationship fragments
AtlanMod Zoo	305	8172	4525	3647
Labeled	62	598	291	307

Table 3.1. UML datasets and their sizes by version

Since the goal of this paper is the translation of specifications into diagrams, each UML class diagram needs to be paired with an English specification. To achieve that goal, we set up a website where we crowdsource the labeling of fragments. The website was hosted on the Heroku cloud platform for a monthly fee.

[1] `https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Zoos`

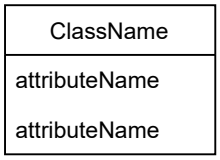


Fig. 3.2. Class fragment

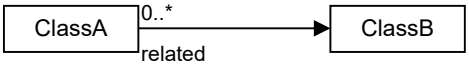


Fig. 3.3. Relationship fragment

Web development is done using Django [2] as the web framework, JavaScript [3] for advanced visualization, and PostgreSQL [4] for database management. The website is made of a landing page and a work page where users contribute labels for the thesis. The work page displays one UML fragment at a time and prevents other users from seeing this fragment so only one label is received per fragment. When a fragment is labeled, the website shows another fragment from the same UML diagram, until all fragments of the same diagram are labeled. On the work page, we also show the full UML class diagram and some examples of labeled fragments, to help users write their English specifications.

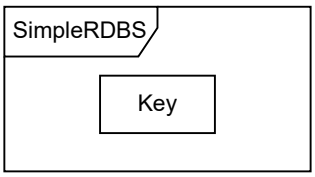
We design each labeling session to be anonymous and as short as possible. The minimum expected time spent per session is 5 minutes. Fragments are presented in the ascending order of complexity of their parent UML diagram. As such, the first fragment comes from the smallest UML diagram. A total of 305 diagrams containing 8172 fragments were presented for labeling.

To find volunteers, we send invitations to different model-driven engineering (MDE, see Section 2.1.3) mailing lists and specific large research groups active in the MDE field. Volunteer participants are mostly university students and faculty members across the world. To ensure that the labeling is done in good faith, we do not offer monetary compensation for participation. However, since participation was low, we did not impose a contribution limit. Because we promised anonymity to all participants, we have no records of how many people took part in our experiment.

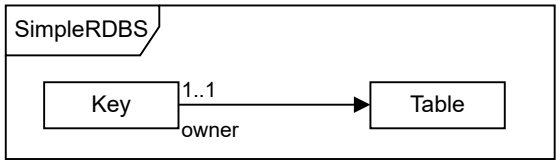
After about two months of crowdsourcing, we received labels for 598 fragments across 62 UML class diagrams. The produced dataset is available on a public repository[5]. To ensure quality, labels are reviewed and less than 5 labels were rejected. We replace the rejected labels by labeling them again ourselves. Figure 3.4 shows example labels. Our criteria for acceptance are not very strict. As long as the specification vaguely describes the associated

[2] `https://www.djangoproject.com/`
[3] `https://www.javascript.com/`
[4] `https://www.postgresql.org/`
[5] `https://github.com/XsongyangX/uml-classes-and-specs`

UML class diagram, it is acceptable. We do not correct English spelling and grammar errors in the labels.



(a) *Key is a class in SimpleRDBMS package*



(b) *A Key is owned by one and only one Table in an RDBMS*

Fig. 3.4. Example labels received by crowdsourcing

We attach some screenshots of the website that volunteers used to label fragments. The website is called "The UML Labeling Initiative". See Figures 3.5, 3.6, A.1, A.2 and A.3.

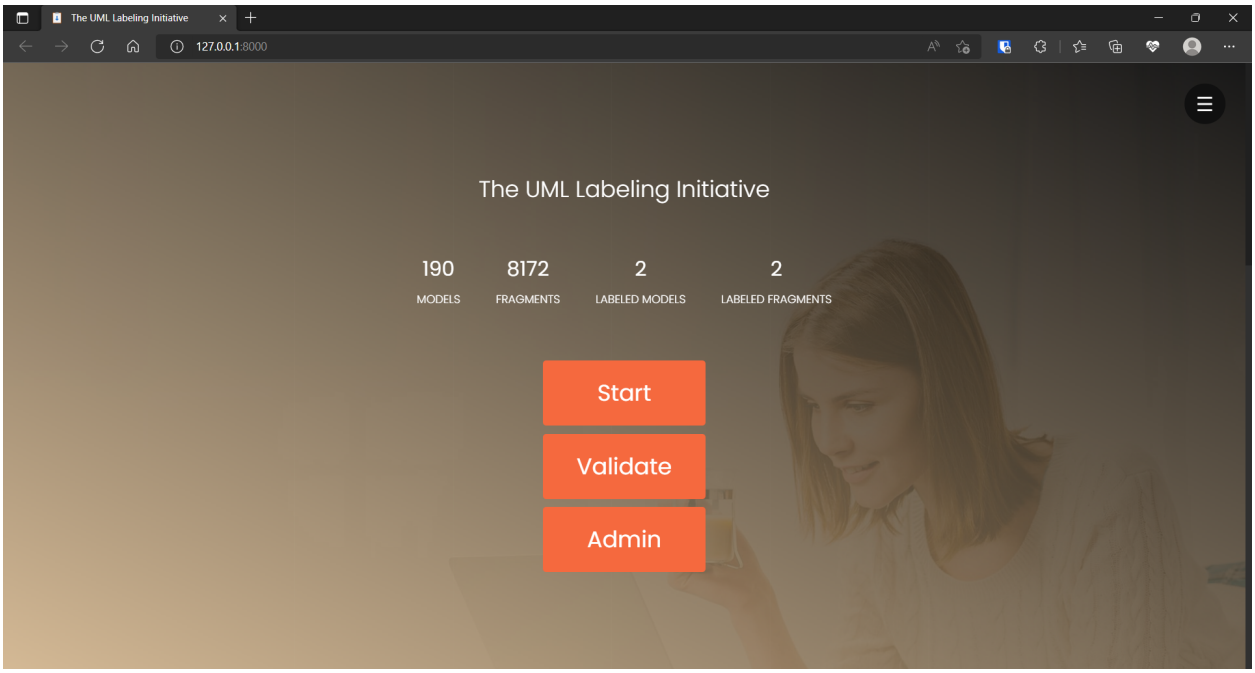


Fig. 3.5. Landing page of the UML Labeling Initiative

3.2. Preprocessing and Fragmentation of Specifications

Preprocessing is the first step after receiving an input specification from the user as shown in Figure 3.1. We substitute pronouns throughout the text, such as *it* and *him*, with their

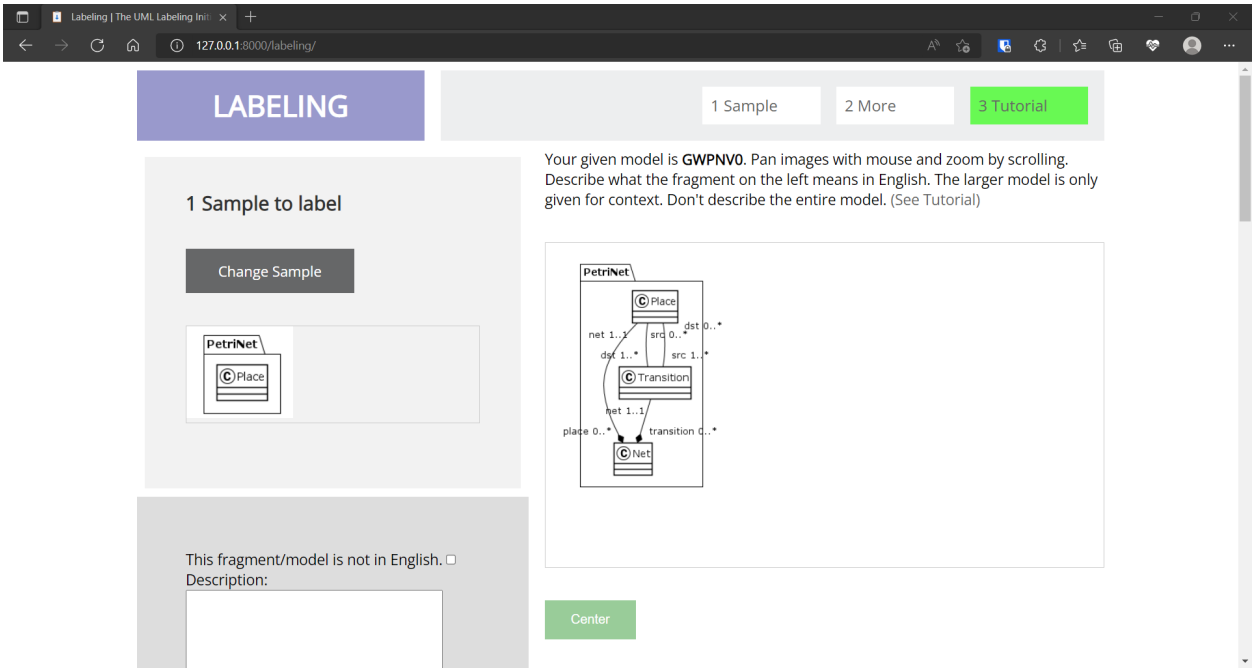


Fig. 3.6. Interface of the UML Labeling Initiative

reference nouns. This is done using *coreferee* [**11**], which is a tool written in Python that performs coreference resolution, including pronoun substitution. The reason for this is to give semantic independence to each sentence in a specification. This way, a sentence has enough information to produce a UML fragment independently. The following sentence is an example.

> «*A course is taught by a teacher. A classroom is assigned to* ***it****.*
> $\Longrightarrow$ *A course is taught by a teacher. A classroom is assigned to* ***a course****.*
> »

The accuracy of *coreferee* for general English text is 81% for all kinds of coreference resolution. We only use *coreferee*'s pronoun substitution. As defined in Section 2.1.11, coreference resolution includes the resolution of generic terms, such as substituting "the country" with "Canada". Though it gives precision, resolving generic terms will make all sentences describe the same entity. Software specifications may make use of generic terms to convey generalizations as a design pattern. Sometimes, the specification intends there to be two separate UML classes called "Country" and "Canada".

Sentence fragmentation is the second step in the runtime operations in Figure 3.1. We split the preprocessed text into individual sentences, using *spaCy* [**3**]. *spaCy* is an NLP library in Python that can be used for various NLP tasks, such as sentence splitting. *spaCy* splits text into sentences by looking at punctuation and special cases like abbreviations. Its decisions are powered by pre-trained statistical models. We use the small English model,

which has a good speed and respectable performance. For instance, in the following example, the first two dots are not considered for splitting the sentences but the third dot is.

> «*An employee has a level of studies, i.e., a degree. An employee is affiliated to a department.*
> $\Longrightarrow$
> s_1*: An employee has a level of studies, i.e., a degree.*
> s_2*: An employee is affiliated to a department.* »

3.3. Sentence Classification

Sentence classification is the third step in the runtime operations of Figure 3.1. Classification provides additional information on the English specification that can be used later to better generate the related UML diagram fragment. Each sentence is classified as describing either a "class" or a "relationship".

The training data for the classifier comes from the dataset described in Section 3.1. Each data point is structured as a pair <English specification, UML fragment> and is assigned a label of a "class" or "relationship" from the moment the dataset was processed from AtlanMod Zoo. The pairing means that the English specification belongs to that specific UML fragment. Our classifier is trained to predict the "class/relationship" label from an English specification. To evaluate the accuracy of the classifier, we use 80% of the data for the training, and the remaining 20% for testing.

When training classifiers on natural language text, we have to select a method to map those sentences into numerical representations. To this end, we experiment with two vectorization methods, count and TF-IDF, which are designed to turn words into vectors (see Section 2.1.10).

Count vectorization looks at the frequency of words in a sentence and creates a vector based on the observed frequency. The dimension of the vector is the size of the vocabulary and the components of the vector represent the frequency of each word. We obtain the vocabulary by running the vectorizer on the entire dataset.

"Term frequency-inverse document frequency" (TF-IDF) also looks at the frequency of words in a sentence to create a vector. A key difference is the penalization of very common words in a document. This allows giving less importance to words like "the".

As for the classification algorithms, we experiment with various algorithms from the *scikit-learn* library [**25**]. We use the default hyperparameter settings for each algorithm. Table 3.2 shows the performance of the algorithms on the test data. It is worth noting that the training takes less than one minute. The accuracies represent the proportion of correctly predicted labels (class or relationship) over the total amount of predictions made.

	Accuracy by chosen vectorizer	
Classifier	TF-IDF	Count
Bernoulli Bayes	**0.87**	0.83
Multinomial Bayes	0.83	0.85
k neighbors	0.82	0.74
Linear SVC	0.88	0.84
SVC	0.88	0.55
ADA	0.85	0.85
Random Forest	0.81	0.70
Logistic Regression	0.86	0.85-0.95

Table 3.2. Accuracy of binary classification of English sentences

Although some classifiers have better accuracy, we pick the Bernoulli Naive Bayes classifier with a TF-IDF vectorizer. Bernoulli Naive Bayes is simple, has a good accuracy that is more stable across training experiments and is generally faster to execute.

Interestingly, Bernoulli Naive Bayes performs better on a TF-IDF vectorizer than the count vectorizer, when it should perform equally well on both in theory. The Bernoulli Naive Bayes classifier uses the Bernoulli distribution, which is a yes-no probabilty distribution. In our case, it means the Bernoulli Naive Bayes looks at whether a certain word is in a sentence, not at how many times it shows up in the sentence. We attribute the difference in performance to the randomized splitting of the dataset when training and testing. Moreover, if a Bernoulli distribution captures enough information to classify well, it seems frequency-based vectorization is not needed.

A given sentence may describe both a UML class and a UML relationship. In that case, we let the classifier make the decision.

3.4. UML Fragment Generation

After classifying each sentence as describing either a "class" or a "relationship", we generate the corresponding UML fragment according to this classification, which is the fourth step in the runtime operations of Figure 3.1.

Using *spaCy*'s small English model [**3**], we define several grammar patterns to match the English sentences. We design the patterns based on the data we collected through crowdsourcing. We broadly group the patterns in Table 3.3. For sentences labeled as class descriptions, we define eight patterns $CP1$ to $CP8$, and for those describing relationships, we define six patterns $RP1$ to $RP6$.

The patterns make use of part-of-speech tagging and dependency analysis. Part-of-speech tagging means assigning each word with its noun/verb/adjective/etc. Dependency analysis looks at the grammatical function of each word in the sentence. For example, a word can

be the subject or the object of the main verb. *spaCy* receives the patterns in Table 3.3 in Semgrex format [**34**] and explores the parse tree.

Pattern	Example of matching sentence
Class Fragments	
CP1: Copula (conjugated verb "to be")	*Key is a class in SimpleRDBMS package*
CP2: "there is"	*There is a place.*
CP3: Compound noun	*Drawing Interchange Format*
CP4: Compound explicit	*Workflow State class*
CP5: "to have"	*a Mesh has a name of type String*
CP6: "class named"	*A class named "Actor".*
CP7: "of package"	*TextualPathExp is part of the package TextualPathExp*
CP8: "and" clauses	*News have titles and links*
Relationship Fragments	
RP1: "to have"	*A MSProject has at least one task.*
RP2: Passive voice	*A news is published on a specific date*
RP3: "composed"	*A node is composed of a label*
RP4: Active voice	*Eclipse plugins may require other plugins*
RP5: Noun "with"	*A table with a caption*
RP6: Copula	*In a Petri Net a Place may be the destination of a Transition*

Table 3.3. Summary of patterns

The patterns in Table 3.3 are the ones we encountered the most in our dataset. There can be as many patterns as possible. The listed patterns are not exhaustive.

Multiple patterns can overlap and as such, the *spaCy* parser produces several parse trees for the same sentence. For example, in the category of class fragments, the patterns *CP3: compound noun* and *CP4: compound explicit* are likely to be both applied at the same time. In this case, we set the *CP4: compound explicit* pattern at a higher priority and discard the parse tree from *CP3: compound noun*.

In general, the priority of patterns in the event of multiple parse trees is based on how specific the pattern is and how much information can be acquired in the parse tree. Hence, for relationship fragments, the patterns for passive voice and active voice are so general that they always yield priority to the other patterns. Table 3.4 shows the priority of each pattern relative to each other, with the top line being the highest priority.

After a pattern and its parse tree have been chosen, we generate a UML fragment using a specific template.

If the classification of the sentence resulted in "class", we generate a UML class fragment consisting of only one class with some possible attributes. For example, the *CP8: and clause* pattern creates a class whose name is the subject noun and whose attributes are the objects

In order of high priority	
Class	**Relationship**
CP1: Copula	RP1: "to have"
CP2: "there is"	RP3: "composed"
CP8: "and" clauses	RP5: Noun "with"
CP5: "to have"	RP2: Passive voice
CP6: "class named"	RP4: Active voice
CP4: Compound explicit	RP6: Copula
CP3: Compound noun	
CP7: "of package"	

Table 3.4. Priority of patterns with the top line being the highest priority

among the "and" clauses. Figure 3.7 shows this result. Above the horizontal arrow is the sentence and below the arrow is the resulting UML class fragment.

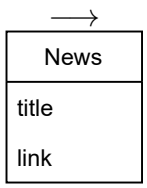


Fig. 3.7. Example of class generation

If the classification of the sentence resulted in "relationship", we generate a UML fragment with two classes and one unidirectional association between them. In the case of a "relationship", we can also extract the multiplicity by looking for keyword expressions like in Table 3.5. Here is an example with the pattern *RP1: to have* in Figure 3.8. Similar to Figure 3.7, the sentence above the arrow is the input. The UML class diagram below the arrow is the output, but this time it contains an association arrow and two classes.

Multiplicity	UML Notation	English expressions
Any number	0..*	"zero or more", "zero or many"
Unique	1..1	"one and only one", "exactly one", "one and only"
One or more	1..*	"one or more", "at least one", "one or several"

Table 3.5. How multiplicities are expressed

We decide to not encode multiplicity expressions directly into the *RP* pattern grammars. Doing so would increase the number of grammars significantly and make the parser harder to implement.

During the creation of UML class fragments, we perform additional processing on the parse tree results, such as lemmatization, noun phrase discovery, and variable naming. These

A MSProject has at least one task.

$\longrightarrow$

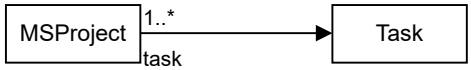


Fig. 3.8. Example of relationship generation

steps are necessary because a UML diagram makes use of words expressed for programming languages rather than English prose, and because the grammar patterns ignore expressions where nouns have complements. Table 3.6 summarizes the additional steps.

Processing method	Examples
Lemmatization	"titles" $\longrightarrow$ "title", "opened" $\longrightarrow$ "open"
Noun phrase discovery	"News have bold titles and url links" $\longrightarrow$ "bold titles", "url links"
Variable naming	Class names: "BoldTitle" Attributes and relationships: "boldTitle"

Table 3.6. Additional processing following parse results

In Table 3.6, lemmatization changes all nouns to singular and verbs to the indicative. Lemmatization respects the variable naming convention of having words in the singular or non-conjugated forms. To take into account the words around a noun, we look at noun phrases in the sentence. A noun phrase is anything that complements a noun. That can be an adjective, another noun, or even an adverb modifying the adjective attached to a noun. All words in a noun phrase are combined together following a variable naming convention, depending on whether the noun phrase is describing a class name or attributes and relationships. These additional steps are all done using *spaCy*.

3.5. Composition of UML Fragments

The last step in the runtime operations of Figure 3.1 is the composition of UML fragments into one UML class diagram. After each sentence is turned into a UML fragment, we produce the final UML diagram by combining the fragments together. Since the merging of general UML class diagrams is NP-hard [**28**], we design an algorithm tailored to our use case. The time complexity of our algorithm is polynomial, more specifically $O((c+r)a^2c)$ where c is the number of classes, r is the number of relationships between classes and a is the number of class attributes.

The composition algorithm takes a greedy approach. Algorithm 1 merges one UML fragment at a time into a larger, work-in-progress UML diagram. When all fragments are used, the work-in-process diagram is the completed UML diagram.

Algorithm 1 Composition algorithm

```
1: model ← previous composition result or any fragment if the composition is just starting
2: f ← incoming fragment
3: if kind(f) = class then
4:     if ∃c ∈ model.classes where c.name = f.name then
5:         if ∃ Attribute-Class conflict then
6:             resolve Attribute-Class conflict according to Figure 3.9
7:         else
8:             merge attributes from f into c with Algorithm 2
9:     else
10:        insert f into model
11: else if kind(f) = relationship then
12:    if ∃ Attribute-Relationship conflict then
13:        resolve Attribute-Relationship conflict according to Figure 3.10
14:    left ← class from which f points
15:    right ← class to which f points
16:    if left ∉ model.classes then
17:        insert left into model
18:    if right ∉ model.classes then
19:        insert right into model
20:    if f's relationship ∈ model.relationships then
21:        do nothing
22:    else
23:        insert f into model
24: return model
```

Algorithm 2 Merge attributes

```
1: class ← recipient UML class
2: a ← incoming attribute
3: if ∃c ∈ class.attributes where c.name = a.name then
4:     if a has a type and c has no type then
5:         replace c by a inside class
6: else
7:     insert c into class.attributes
8: return class
```

During composition, fragments may present contradicting information to the model in progress. We identify two situations for this. The first is an Attribute-Class conflict and the second is an Attribute-Relationship conflict (Figure 3.10). In both situations, the resolution involves removing attributes to create a new class or a new relationship. We favor having many smaller classes and relationships, instead of a few very big classes.

An Attribute-Class conflict arises when the UML class diagram in progress contains an attribute with a name identical to the name of a class from a class fragment. We resolve this

conflict by removing the attribute from the diagram in progress, inserting the class fragment into the larger UML diagram, and creating a new relationship from the class that previously contained the attribute to the inserted class. This relationship has the name of the attribute as its name and a multiplicity of zero-to-many. See an example in Figure 3.9.

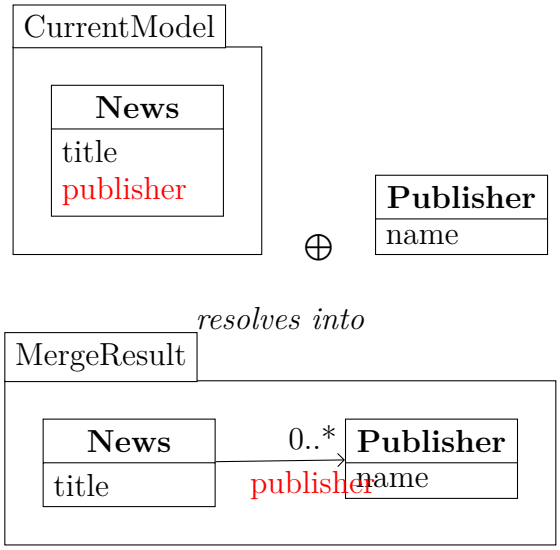


Fig. 3.9. Attribute-Class conflict and its resolution

An Attribute-Relationship conflict arises when the UML class diagram in progress contains an attribute a whose name is identical to the name of the relationship inside the incoming relationship fragment. Let A be the class of this attribute a in the diagram in progress. Let C be the class that is the source of the unidirectional association inside the relationship fragment. Let D be the class that is the destination of the unidirectional association inside the relationship fragment. If the names of A and C are not the same, this is not a conflict and we proceed with a standard insertion. Otherwise, the resolution starts by removing the attribute a from A. Then we merge the attributes of A and C. The relationship from C to D is now from the attribute merge result $A \bigoplus C$ to D.

Lines 16 and 18 in the composition algorithm (Algorithm 1) make use of "relationship" equality. We define two relationships to be equal if the classes they are related to have the same name and if the name of the relationship is the same after processing. This implies that multiplicity is ignored when assessing equality.

Finally, this entire pipeline produces one UML class diagram from the received input. We compile the result into an image using a compiler called *plantuml* [**26**].

The proposed algorithm is not the only way of composing fragments together.

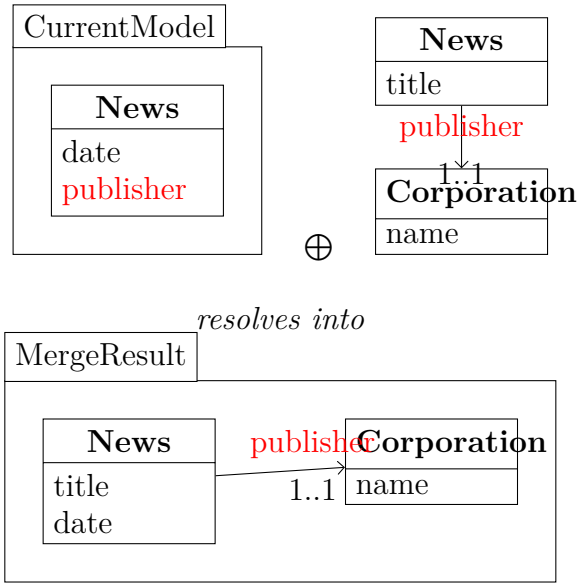


Fig. 3.10. Attribute-Relationship conflict and its resolution

Chapter 4

Evaluation

Overview

In this chapter, we evaluate our approach and discuss the results. We create a testing framework from the data of Section 3.1 using metrics of varying strictness. We obtain low performances and explain their implications for our approach. Lastly, we list out some threats to validity.

4.1. Setup

To test the performance of our approach, we use the dataset we created through crowdsourcing in Section 3.1. We first group all the English specifications for fragments by the UML model they originated from. This creates 62 testing samples. For example, the following grouped specification corresponds to the UML class diagram shown in Figure 4.1.

> «*Drawing Interchange Format. a Drawing Interchange model may have multiple meshes. a Mesh has a name of type String. a Mesh may have any number of points. a point maps to only one Mesh. a point has a name of type String and coordinates X and Z of type Double.* »

4.2. Evaluation Metrics

To evaluate the accuracy of our approach, we use comparative metrics. We design three levels of strictness for comparing the diagrams generated by our tool and the ground truth of the dataset. We assume there is only one good UML class diagram per input specification, because our dataset has only one correct UML class diagram per English specification.

First, we have **exact matching**, which is the most strict comparison. Under exact matching, we look at how many classes and relationships from the ground truth are present in the generated diagrams. A ground truth class is considered present in the prediction if there is a class in the generated diagram that has an identical name and identical attributes.

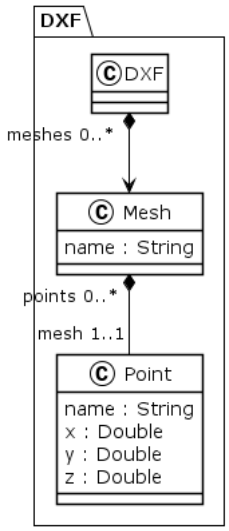


Fig. 4.1. An original UML class diagram from the AtlanMod Zoo

A ground truth relationship is considered "present" in the prediction if both of the classes attached to it are present and if there is a relationship between these two classes with the same name and multiplicity in the predicted output. Each UML diagram under evaluation outputs precision, recall, and f-1 score.

Second, we have **relaxed matching**, which is a weaker form of exact matching. In relaxed matching, we still look at how many ground truth classes and relationships are present in the generated diagram. However, a ground truth class is considered "present" if there is a predicted class with the same name. We don't look at attributes anymore. Similarly, ground truth relationships are considered present in the prediction in the same way as exact matching, except that multiplicities are ignored.

Third, we have **general matching**. This is the most lenient matching criterion. In general matching, classes are still evaluated like in relaxed matching. Relationships, on the other hand, are evaluated collectively instead of individually. We look at a diagram's graph connectivity and compare this connectivity to the ground truth diagram's connectivity. This comparative metric ignores class names, the orientation of relationships, and the name of relationships. As such, general matching ignores semantics.

To compare two diagrams' connectivity, we use the technique of eigenvector similarity [**13**]. In short, this technique looks at the eigenvalues of the Laplacian matrices of the two undirected graphs. If the distance between the most prominent eigenvalues is small, the two graphs are similarly connected. Distances are in the range $[0, \infty)$. We apply a mapping to normalize the distance into a score in the interval (0,1], where 0 means no similarity at all and 1 means the two diagrams are identically connected. The mapping is $f(x) = 2(1-\sigma(x))$, where $x \in [0, \infty)$ is the distance and $\sigma(x)$ is the sigmoid function. We plot the mapping

function in Figure 4.2. A perfect connectivity score does not mean the two diagrams are identical.

Fig. 4.2. Mapping positive reals to the [0,1] interval using $f(x) = 2(1 - \sigma(x))$

To complement connectivity, we add a size difference score to the general matching metric. The size difference is evaluated by computing $||s_1 - s_2||$, where $s_i = normalize$([number of nodes, number of edges]) of graph i. The norm is the Pythagorean distance, and the graphs are the two undirected graphs used in the eigenvector similarity calculation. The norm ranges from 0 to $\sqrt{2}$, with 0 being the best score. We apply the mapping $g(x) = 1 - x/\sqrt{2}$ to make the score fall in the interval [0,1] with 0 being the worst score and 1 being the best score. This means 0 is attributed to graphs with vastly different sizes, and 1 is attributed to graphs with similar sizes. A score of 1 means the size vectors s_i are oriented closely, not that the graphs have the same sizes. To get a better grasp, we look at the precision and recall results for class generation and the connectivity similarity.

4.3. Results

After generating 62 candidate UML class diagrams from English specifications, we use an automated script to compare the predictions with the ground truth. Each predicted diagram is compared with its ground truth counterpart in the dataset. We take the average of the 62 results for all metrics and present it in Tables 4.1 and 4.2.

Table 4.1 shows the results for class generation. Under exact matching, the precision is 17%, the recall is 25% and the f-1 score is 20%. This low performance means that most predictions are not identical to the UML diagrams of the dataset we created in Section 3.1. Differences can arise when the names of the classes and attributes are not identical character

Metric	Precision	Recall	F-1 score
Exact matching	0.171	0.251	0.200
Relaxed matching	0.355	0.506	0.409
General matching	0.355	0.506	0.409

Table 4.1. Performance of generating classes from English specification

Metric	Connectivity similarity	Size difference
General matching	0.639	0.673

Table 4.2. Performance of generating relationships from English specification

by character. In addition, a match is considered negative if attributes do not have the same type. Here, recall is higher than precision. That means our approach generates more classes than needed, many of which are not found in the ground truth data.

In Table 4.1, the relaxed matching metric outputs a precision of 35%, a recall of 50%, and an f-1 score of 40%. As expected, the performance is higher than exact matching because relaxed matching ignores class attributes. Similar to exact matching, a recall higher than precision means that the predictions have more classes than necessary. But in this metric, it seems that much more ground truth classes are found in the prediction. If we take the difference between the two metrics, we get a gap of 18% in precision, 25% in recall, and 20% in f-1 score. This significant gap suggests that our attribute extraction has room for improvement.

General matching for classes is defined in the same way as relaxed matching. As such, there is no difference in performance.

Figure 4.3 is an example of our approach generating too many classes and containing too little attributes. We notice that the package name is generated as its own class. The attributes of "Point" in the original diagram are missing in the generation. Given that these attributes are "x", "y" and "z", the English grammar parser might have trouble recognizing them.

For relationships, Table 4.2 shows the approach's performance, which only consists of the general matching metric. Other matching metrics offer inconclusive results because a positive result requires the classes to be all positive too, which is not possible when class prediction performance is below 50%. We achieve a connectivity similarity of 63% and a size difference of 67%. A connectivity score of 63% implies the nodes and edges of the graph are connected quite differently. Some classes have too little or too many relationships. A size difference score of 67% means that diagrams have a noticeable size difference since precision is also low.

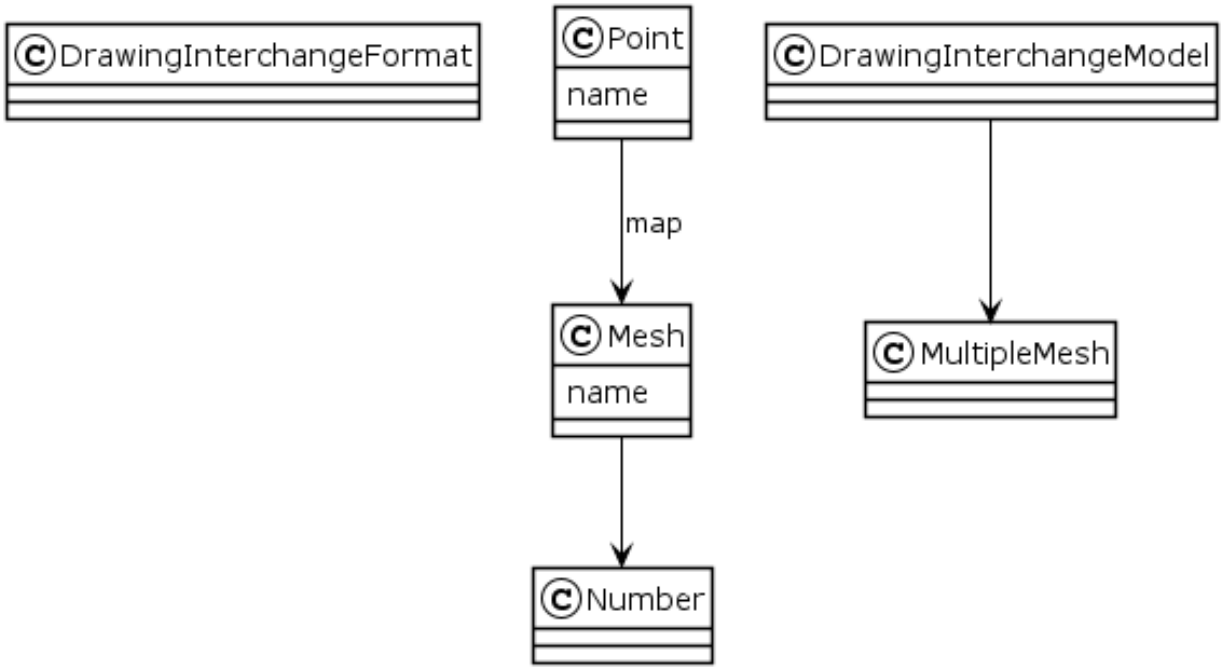


Fig. 4.3. The generated UML class diagram from the original (Figure 4.1)

4.4. Discussion

The evaluation results are not great. The performances are below 50%, which makes our approach not suited for practical use. We explore several reasons for this and the meaning of low results in the challenge of extracting UML class diagrams from natural language specifications.

Since the precision for class generation is low at 17% for exact matching and 35% for relaxed matching, it might be caused by too many classes in the prediction that can be traced to synonyms in the specification. English specifications can contain synonyms and different wordings for the same idea. If the user decides to use two different terms for the same concept in a specification, they might have wanted two different classes, or the user might have wanted a specification that is more interesting for humans to read. This ambiguity cannot be resolved without user feedback. However, given our approach generates too many classes on average, a more aggressive merging during the composition step (Section 3.5) would be beneficial.

Figure 4.4 shows an example where the generated diagram contains synonyms. The synonyms are "Finiteautomaton", "FiniteautomatonClass" and "Automatoon". The multiword class names differ only by how specific they are, while "Automatoon" differs by spelling. Our method cannot mitigate user-provided typos.

While our approach generates too many classes, recall for class generation is still too low at 25% for exact matching and 50% for relaxed matching. This means there are elements

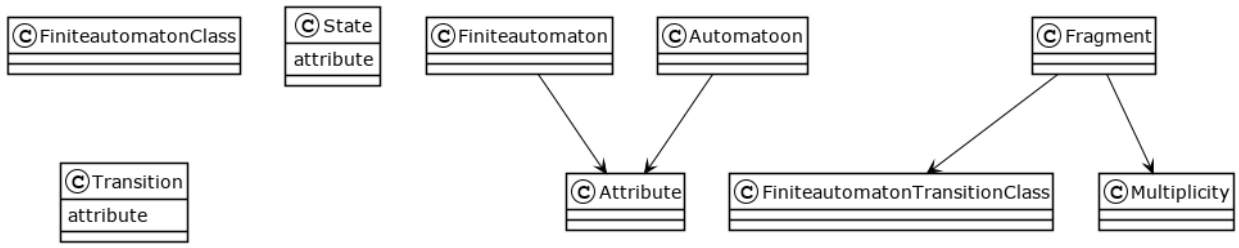


Fig. 4.4. A generated UML class diagram with many synonymous class names

in the ground truth UML class diagram that are not extracted from English specifications. This can be improved by adding more patterns to the rule set in Section 3.4. Moreover, a better noun phrase extraction mechanism can extract more class names and attributes from the text. Currently, our method works well with noun phrases that are one word or two words long. Noun phrases longer than two words require a more sophisticated extraction process. We used *spaCy*'s default noun phrase detector, but exploring the dependency parse tree ourselves directly might be a better idea.

For example, the following specification cannot be parsed and understood easily by our approach. Relative clauses in sentence are hard to derive information from. A possible way to solve this is to turn all relative clauses into independent sentences in the preprocessing stage.

> «*A zoo that has animals and different pricing rates. Zoo handlers which whose work schedule depends on the day.* »

A low performance signals that our approach has limits. Since we incorporate several statistical components with their own imperfect accuracy in our pipeline, there is an upper ceiling of performance we cannot exceed. Our performance cannot be better than the performances of these components. In fact, if we assume all components have an equal influence on the output, we have an expected upper limit of accuracy of 0.63 as seen in Table 4.3. One way of reducing the effect of compounding errors is to introduce a retroactive step. In our case, that step is the composition algorithm's attempt to resolve conflicts in Algorithm 1.

Statistical component	Accuracy
Pronoun resolution (*coreferee*)	0.81
Grammar analysis (*spaCy*)	0.90
Binary classification	0.87
Combined product	**0.63**

Table 4.3. Accuracies of each statistical component in our approach

4.4.1. Evaluation metrics

The definition of general matching makes use of two new metrics. The connectivity similarity uses eigenvector similarity that produces a positive real distance. Mapping real numbers to a finite interval results in distortions. Our mapping $f(x) = 2(1 - \sigma(x))$ makes distances above 3 very close to 0 in percentage scores, as seen on Figure 4.2. This implies that very large positive real distances yield a very close percentage score in the [0,1] interval. This might be problematic because percentages are often perceived linearly, and close percentages should intuitively represent similar performances.

The other problematic general matching metric is the size difference score. The definition is $||s_1 - s_2||$ where $s_i = normalize([|V_i|, |E_i|])$ for graph $G_i = (V_i, E_i)$. The size difference score cannot distinguish between graphs of sizes that are multiples of each other. For example, if a graph has 2 nodes and one edge, then the size difference score would be the same when computed with a graph with 4 nodes and 2 edges.

4.4.2. Conclusion

To conclude the evaluation, we can mention that this work is an initial attempt to solve the problem of diagram generation from natural-language specification by combining machine learning and natural language parsing patterns. Although the overall performance in Table 3.2 and Table 4.2 is limited, this work allows us to identify the improvement possibilities. Additionally, the approach itself can serve as a baseline for future research on that problem, and the dataset can be used as a benchmark for the same problem.

4.5. Threats to Validity

4.5.1. Dataset

Although we rejected bad labels during the creation of the dataset, some volunteers provided specifications with questionable semantics and spelling errors. We kept those labels because we want our approach to operate under imperfect conditions. Our approach cannot deal with spelling errors and confusing specifications. Each spelling mistake creates an extra UML class or relationship that should not exist. And if the specification is unclear, then the generated diagram is also unclear. A future work could address the spelling mistakes by adopting a spelling correction preprocessing procedure using edit similarity. Uncertain words could be flagged and what they end up generating could be merged during composition.

Due to a lack of sufficient data, we did not set aside unseen data for the evaluation. The evaluation uses the entire dataset for testing. Despite the classifier of Section 3.3 splitting the data into 80-20, 80% of the evaluation data has been seen during the training of the

classifier. We believe this bias to be minimal because the classification is an intermediate step.

4.5.2. Language model used for fragment generation

To speed up the research process, we used the smallest NLP English model on *spaCy*. The accuracy is known to be about 90%. As a consequence, the English model is less accurate when segmenting the text into sentences, performing dependency parsing (definition provided in Section 2.1.8), and discovering noun phrases in Section 3.4. Each error on the part of the language model makes one UML fragment wrong.

For pronoun substitution, we used *coreferee* which has an overall accuracy of 81%. This library is the least accurate among all third-party libraries. If a pronoun is wrongly substituted, our approach generates a very different UML model.

4.5.3. Composition algorithm

In the composition algorithm (Algorithm 1), the merging of the UML class diagram with fragments is treated in a non-commutative fashion. In other words, the pseudo-code only addresses the conflicts when it is an Attribute-Class (Figure 3.9) and not Class-Attribute. A similar situation is happening for the Attribute-Relationship conflict. If the conflicting relationship is already inside the model in progress, the algorithm will not flag that as an Attribute-Relationship conflict and it will therefore not resolve it. An improved version of the algorithm should treat the merging of the UML class diagram and the fragment in two directions, i.e., in a commutative way. This would increase the performance of our approach.

Chapter 5

Conclusion

In this thesis, we propose an automated approach to extract UML class diagrams from English specifications. The approach uses machine learning and pattern-based techniques. Machine learning is used in the form of a binary classifier that labels sentences as either describing a class or a relationship. The pattern-based techniques are handwritten grammar rules to parse English sentences. In this approach, we fragment the English input into sentences, generate UML class diagram fragments from them, and combine all the fragments together into a final result.

To develop our tool, we first create a dataset of UML diagrams paired with English specifications. The specifications are produced by a crowdsourcing initiative. The resulting dataset, although small, is enough to train the classifier and evaluate our approach.

We define three evaluation metrics of varying strictness to test our approach's accuracy in generating classes and relationships from an English specification. The results for classes are 17% precision and 25% recall for exact matching, the strictest metric. The results for relationships are a connectivity similarity of 63% and a size difference of 67%.

The correctness of the produced diagrams is limited. However, these results are in part explained by the imprecision of the NLP tools we used. Using more sophisticated NLP tools will help to improve these results. In addition, more grammar patterns can be added in Section 3.4 and an improved version of the composition algorithm will reduce irrelevant classes.

From a broader perspective, our research lays the work for a consistent quantitative evaluation framework with our approach being the baseline and with the dataset and metrics being the testing framework. From the novelty perspective, we explore intermediate machine learning steps to simplify a mostly rule-based approach. Furthermore, our approach uses a divide-and-conquer strategy when fragmenting diagrams and text and when composing them back together.

In the future, a more complex pattern system can improve the performance of our approach. Currently, we only use a single rule to generate a UML fragment, but if several rules contribute together, the performance can increase. The composition algorithm can also be improved, such as by considering a confidence score in each fragment. Furthermore, inheritance can be generated as a new type of relationship by adding more grammar patterns. Finally, we can generalize our approach to handle other types of UML diagrams, in particular behavioral ones.

References

[1] Esra A Abdelnabi, Abdelsalam M Maatuk, Tawfig M Abdelaziz, and Salwa M Elakeili. Generating UML class diagram using NLP techniques and heuristic rules. In *2020 20th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, pages 277–282. IEEE, 2020.

[2] Esra A. Abdelnabi, Abdelsalam M. Maatuk, and Mohammed Hagal. Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques. In *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, pages 288–293, 2021.

[3] Explosion AI. spaCy: Industrial-Strength Natural Language Processing, 2016.

[4] V. S. Alagar and K. Periyasamy. *The Role of Specification*, pages 3–22. Springer London, London, 2011.

[5] Robert Balzer and Neil Goldman. Principles of good software specification and their implications for specification languages. In *Proceedings of the May 4-7, 1981, national computer conference*, pages 393–400, 1981.

[6] Roger W Brown. Linguistic determinism and the part of speech. *The Journal of Abnormal and Social Psychology*, 55(1):1, 1957.

[7] Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In *International Conference on Model Driven Engineering Languages and Systems*, pages 54–68. Springer, 2005.

[8] K. R. Chowdhary. *Natural Language Processing*, pages 603–649. Springer India, New Delhi, 2020.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[11] Richard Paul Hudson. Coreferee: Coreference resolution for multiple languages, 2021.

[12] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 22–32, 2002.

[13] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. Algorithms for Graph Similarity and Subgraph Matching. page 15–16, 2011.

[14] Urszula Krzeszewska, Aneta Poniszewska-Marańda, and Joanna Ochelska-Mierzejewska. Systematic comparison of vectorization methods in classification context. *Applied Sciences*, 12(10), 2022.

[15] Ling Liu and M Tamer Özsu. *Encyclopedia of database systems*, volume 6. Springer, 2009.

[16] Angel R Martinez. Part-of-speech tagging. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(1):107–113, 2012.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, 2013.

[18] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[19] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical software engineering*, 18(1):89–116, 2013.

[20] Priyanka More and Rashmi Phalnikar. Generating UML diagrams from natural language specifications. *International Journal of Applied Information Systems*, 1(8):19–23, 2012.

[21] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 09 2011.

[22] Iftikhar Azim Niaz and Jiro Tanaka. Code generation from uml statecharts. In *Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey*, pages 315–321, 2003.

[23] Jill Nicola, Mark Mayfield, and Mike Abney. *Streamlined object modeling: Patterns, rules, and implementation*. Pearson Education, 2001.

[24] Joakim Nivre. Dependency Parsing. *Language and Linguistics Compass*, 4(3):138–152, 2010.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[26] PlantUML. Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams., 2021.

[27] Joël Plisson, Nada Lavrac, Dunja Mladenic, et al. A rule based approach to word lemmatization. In *Proceedings of IS*, volume 3, pages 83–86, 2004.

[28] Julia Rubin and Marsha Chechik. N-way model merging. In *proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 301–311, 2013.

[29] Rijul Saini, Gunter Mussbacher, Jin LC Guo, and Jörg Kienzle. DoMoBOT: a bot for automated and interactive domain modelling. In *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings*, pages 1–10, 2020.

[30] Richard Socher, Yoshua Bengio, and Christopher D. Manning. Deep Learning for NLP (without Magic). In *Tutorial Abstracts of ACL 2012*, ACL '12, page 5, USA, 2012. Association for Computational Linguistics.

[31] Rhea Sukthanker, Soujanya Poria, Erik Cambria, and Ramkumar Thirunavukarasu. Anaphora and coreference resolution: A review. *Information Fusion*, 59:139–162, 2020.

[32] Marcin Szlenk. Formal semantics and reasoning about uml class diagram. In *2006 International Conference on Dependability of Computer Systems*, pages 51–59. IEEE, 2006.

[33] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys (CSUR)*, 28(3):438–479, 1996.

[34] Fabio Tamburini. Semgrex-Plus: a tool for automatic dependency-graph rewriting. In *Proceedings of the Fourth International Conference on Dependency Linguistics (Depling 2017)*, pages 248–254, 2017.

[35] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2013.

[36] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *2010 2nd IEEE International conference on information management and engineering*, pages 352–356. IEEE, 2010.

Appendix A

The UML Labeling Initiative

In Section 3.1, we create a website for volunteers to contribute labels for our dataset. The following figures contain more screenshots of the website.

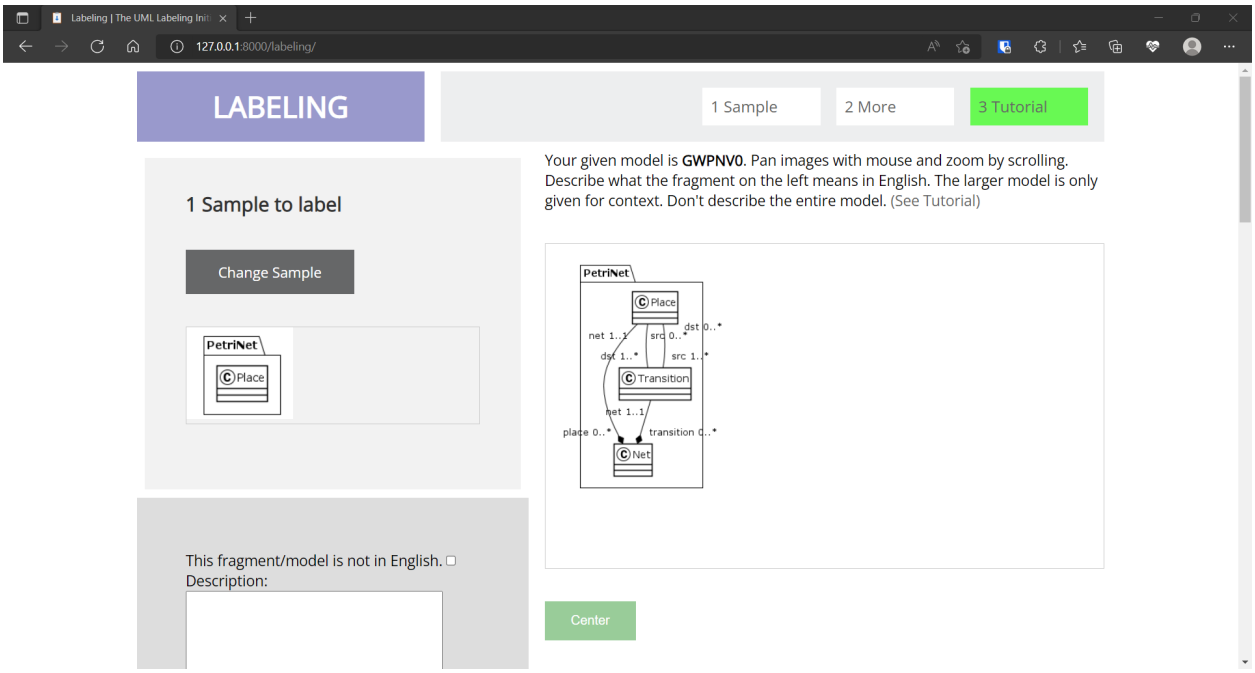


Fig. A.1. The website presents one fragment from one model.

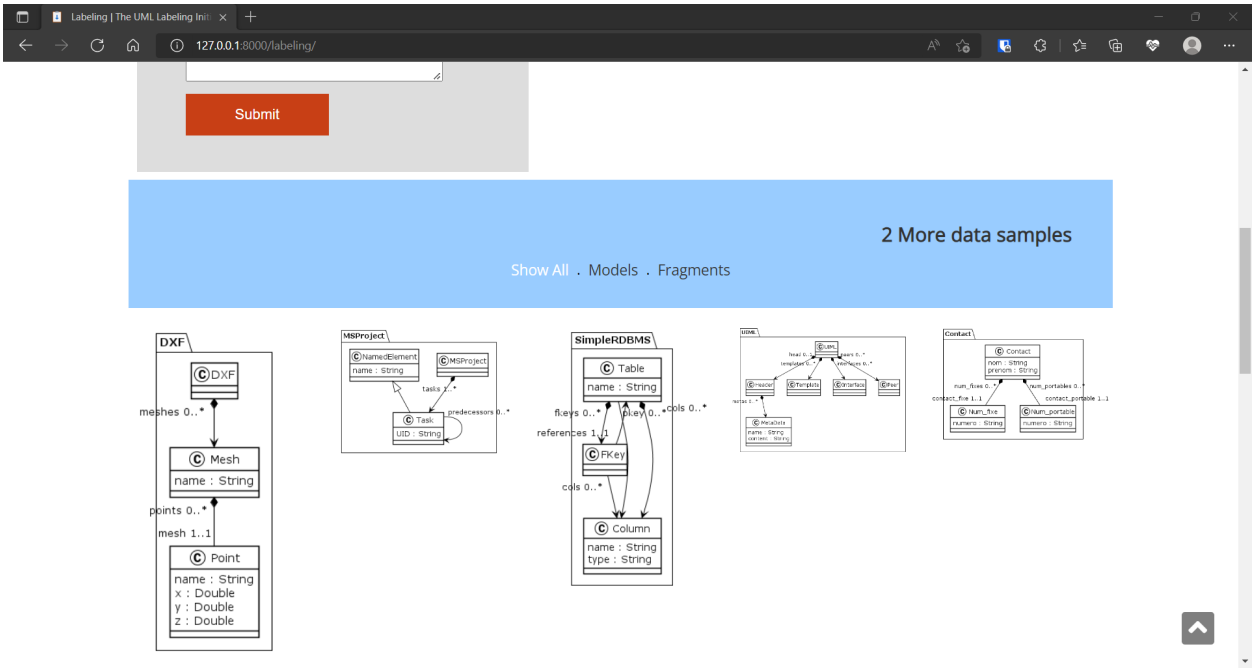


Fig. A.2. Users can choose other fragments and models.

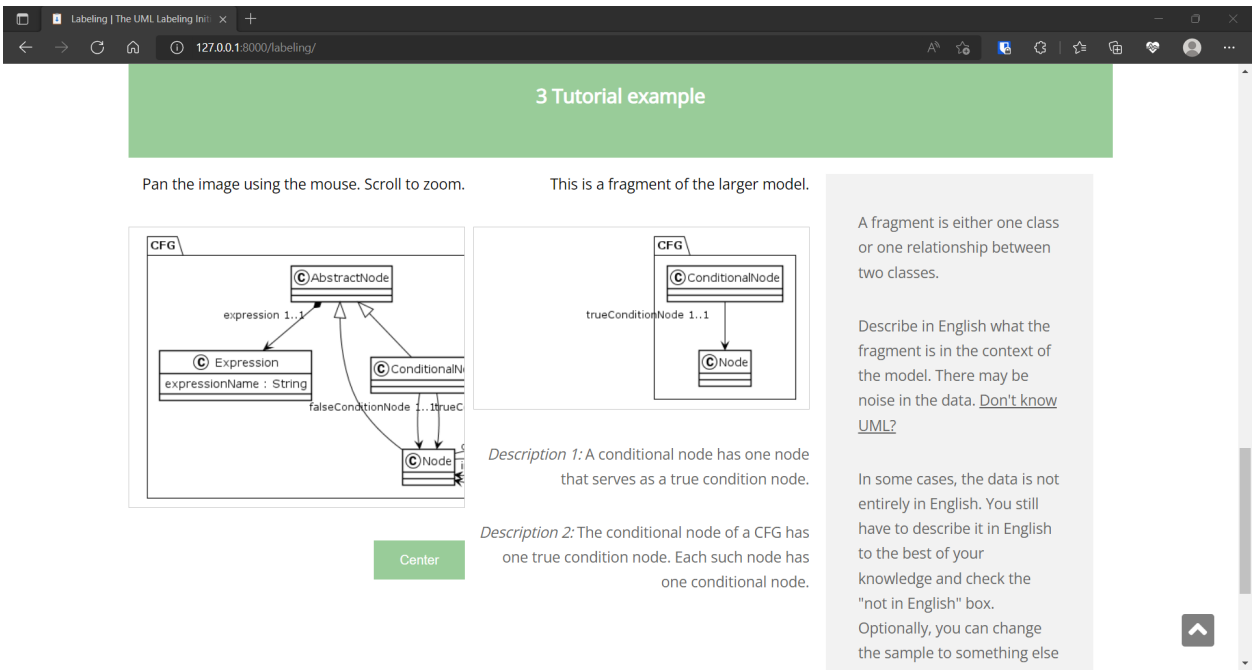


Fig. A.3. Examples and explanations for users on how to label