An NLP-based architecture for the autocompletion of partial domain models

Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, Jordi Cabot

▶ To cite this version:

Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, Jordi Cabot. An NLP-based architecture for the autocompletion of partial domain models. 33rd International Conference on Advanced Information Systems Engineering (CAiSE'21), Jun 2021, Melbourne, Australia. hal-03010872

HAL Id: hal-03010872
https://hal.science/hal-03010872

Submitted on 17 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A NLP-based architecture for the autocompletion of partial domain models

Exploratory paper

Loli Burgueño[1], Robert Clarisó[2], Shuai Li[2], Sébastien Gérard[2], and Jordi Cabot[3]

[1] Open University of Catalonia, Av. Tibidabo, 39-43. Barcelona, Spain
{lburguenoc,rclariso}@uoc.edu
[2] Institut LIST, CEA, Université Paris-Saclay, Avenue de la Vauve. Palaiseau, France
{Shuai.LI,Sebastien.GERARD}@cea.fr
[3] ICREA, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. Domain models capture the key concepts and relationships of a business domain. Typically, domain models are manually defined by software designers in the initial phases of a software development cycle, based on their interactions with the client and their own domain expertise. Given the key role of domain models in the quality of the final system, it is important that they properly reflect the reality of the business.

To facilitate the definition of domain models and improve their quality, we propose to move towards a more assisted domain modeling building process where an NLP-based assistant will provide autocomplete suggestions for the partial model under construction based on the automatic analysis of the textual information available for the project (contextual knowledge) and/or its related business domain (general knowledge). The process will also take into account the feedback collected from the designer's interaction with the assistant. We have developed a proof-of-concept tool and have performed a preliminary evaluation that shows promising results.

Keywords: domain model · autocomplete · modeling recommendations · assistant · natural language processing.

1 Introduction

Domain modeling is the activity in which informal descriptions of a (business) domain are translated into a structured and unambiguous representation using a concrete (formal) notation. Domain models, also known as conceptual schemas [29], are built as part of a software development project to abstract the key concepts of the domain relevant for the project, leaving out superfluous details.

The use of domain models is widely extended and there is a broad variety of languages (UML, DSLs, ER, etc.), tools and methods [12] that promote and facilitate their creation and manipulation. Nevertheless, they are typically created by hand during the analysis and design phases of software development, making their definition a crucial (but also time-consuming) task in the development life-cycle. On the other hand, the knowledge to be used as input to define such

domain models is already (partially) captured in textual format in manuals, requirement documents, technical reports, transcripts of interviews, etc. provided by the different stakeholders in the project.

We believe we could exploit this information to assist designers defining domain models. In software development, autocompletion has been heavily studied for years. Mature features such as code autocompletion are integrated by default in IDEs and numerous benefits like faster coding, error prevention and the discovery of new language elements have been proven. Similarly, we propose model completion as a new feature for a future generation of modeling/design tools (i.e., intelligent modeling assistants [28]) that could significantly improve the domain modeling task.

A couple of commercial low-code platforms [30,26] and research efforts [35,15] are exploring model autocompletion but using other knowledge sources or techniques, e.g., the analysis of a collection of previously developed models from where patterns are extracted or ontologies [8]. However, most companies do not have enough models to obtain meaningful results from the former, while the latter limits its suggestions to general knowledge sources. We believe that we can complement these approaches with autocompletions derived from contextual information in natural language documents. Other approaches [1] have leveraged textual information in general data sources like Wikipedia to provide model suggestions. In contrast, in this paper, we propose combining information from different textual sources: documents generated around the project and general data sources (which include basic information that is omitted from the previous ones as it is supposed to be common knowledge in that community). Moreover, we also consider historic information about previously accepted or rejected suggestions.

More specifically, our goal is to assist the software designer by generating potential new model elements to add to the partial model she is already authoring. We believe this is more realistic than trying to generate full models out of the requirements documents in a fully automated way. In this exploratory paper, we propose a configurable framework that follows an iterative approach to help in the modeling process. It uses Natural Language Processing (NLP) techniques for the creation of word embeddings from text documents together with additional NLP tools for the morphological analysis and lemmatization of words. With this NLP support, we have designed a model recommendation engine that queries the NLP models and historical data about previous suggestions accepted or rejected by the designer and builds and suggests potential new domain model elements to add to the ongoing working domain model. Our first experiments show the potential of this line of work.

The rest of the paper is structured as follows. Sect. 2 describes our NLP-based architecture for model autocompletion. Sect. 3 describes the implementation details and Sect. 4 assesses the feasibility of the approach over an industrial case study. Section 5 presents the related work and Sect. 6, we conclude our work.

2 Approach

Our proposal aims to assist designers while they build their domain models. Given a partial domain model, our system is able to propose new model elements that seem relevant to the model-under-construction but are still missing. To provide meaningful suggestions, it relies on knowledge extracted out of textual documents. Two kinds of knowledge/sources are considered: *general*[4] documents and *contextual* (all the specific information that we collect about the project) documents. We do not require these documents to follow any specific template.

General and contextual knowledge complement each other. The need for contextual knowledge is obvious and intuitive: designers appreciate suggestions coming from documents directly related to the project they are modeling. General knowledge is needed when there is no contextual knowledge or this is not enough to provide all meaningful suggestions (i.e., it may not cover all the aspects that have to be described in the domain model as some textual specifications omit aspects considered to be commonly understood by all parties). For instance, project documents may never explicitly state that users have a name since it is common sense and both concepts go hand-by-hand. Thus, general sources of knowledge fill the gaps in contextual knowledge and make this implicit knowledge explicit. Leveraging both types of knowledge to provide model autocomplete suggestions to the designer would significantly improve the quality and completeness of the specified domain models. As most common knowledge sources are available as some type of text documents (this is specially true for the contextual knowledge, embedded in the myriad of documents created during the initial discussions on the scope and features of any software project), we propose to use state-of-the-art NLP techniques to leverage this textual-based knowledge sources.

Methods such as GloVE [31], word2vec [27], FastText [18], BERT [10] and GPT-3 [4] create *word embeddings* (i.e., vectorial representations of words) that preserve certain semantic relationships among the words and about the context in which they usually appear. For instance, a NLP model[5] trained with a general knowledge corpus is able to tell us that the concepts *plane* and *airport* are more closely related than *plane* and *cat* because they appear more frequently together. For example, the Stanford NLP Group's pretrained GloVe model with the Wikipedia corpus estimates that the relatedness (measured as the euclidean distance between vectors) between *plane* and *airport* is 6.94, while the distance between *plane* and *cat* is 9.04. Relatedness is measured by the frequency in which words appear closely together in a corpus of text. Apart from giving a quantifiable measure of relatedness between words, once an NLP model is trained, it enables

[4] According to the Cambridge dictionary: "information on many different subjects that you collect gradually, from reading, television, etc., rather than detailed information on subjects that you have studied formally".

[5] Note that "NLP model" and "domain model" do not refer to the same type of model at all. In the NLP field, a model is the result of analyzing the textual corpus of data (it could be a trained neural network, a statistical model,...). To avoid confusion, in this work, each time we refer to a NLP model, we always refer to it as "NLP model" and never as "model" alone.

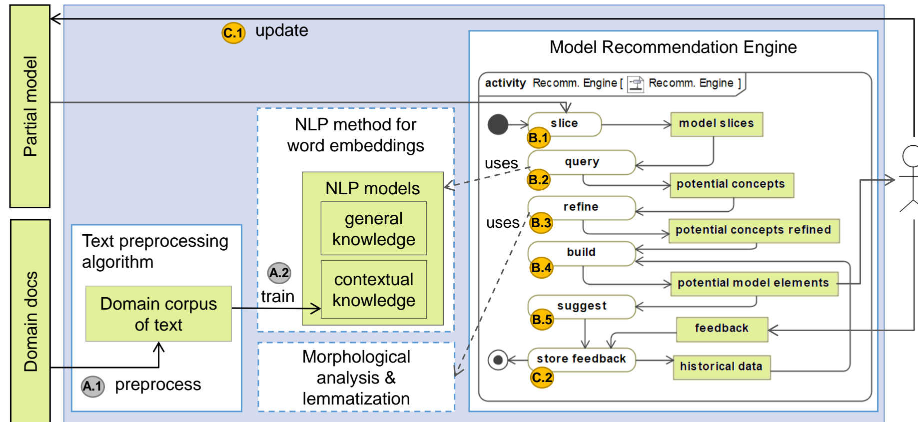


Fig. 1: Autocompletion Framework and Process. *Legend:* Green boxes are data. White boxes are software artifacts. Dotted lines denote already implemented software that we reuse. Solid lines are our contribution/implementation.

us to make queries to obtain an ordered list with the closest words to a given word or set of words. This latter functionality is the one we use in our approach. Another advantage of these techniques is that they are able to deal with text documents regardless of whether they contain structured or unstructured data.

Our framework uses the lexical and semantic information provided by NLP learning algorithms and tools, together with the current state of the partial model and the historical data stored about the designer's interaction with the framework. As output, it provides recommendations for new model elements (classes, attributes and relationships). The main components of our configurable architecture as well as the process that it follows to generate autocompletion suggestions are depicted in Figure 1. The logic of the algorithm implemented for the recommendation engine is depicted using an UML Activity Diagram. We describe our framework architecture as well as all its steps in detail in the following, while Section 3 on tool support provides further technical details.

2.1 Step A: Initialization

Our process starts by preprocessing all the available documentation about the project to use it as input for the NLP training process. This step provides a corpus of text that satisfies the requirements imposed by the NLP algorithm chosen to create the NLP models, e.g., a single text file that contains words separated by spaces. For most NLP algorithms, this step consists of the basic NLP pipeline: tokenization, splitting, and stop-word removal.

Once all the natural language text has been preprocessed (i.e., the domain corpus is available), the NLP contextual model is trained. Note that we could use any of the NLP language encoding/embeddings alternatives mentioned before.

Instead, we do not train a NLP model for the general knowledge every time. Due to the availability of NLP models trained on very large text corpora of general

knowledge data (such as Twitter, Wikipedia or Google News[6]), we propose to reuse them. Therefore, neither *Step A.1* nor *A.2* apply to the general knowledge. Nevertheless, if desired, the use of a pretrained model could be easily replaced by collecting general knowledge documents and executing *Steps A.1* and *A.2* with them.

2.2 Step B: Suggestion Generation

Step B.1. Model Slicing The input to this step is a partial domain model (e.g., a UML model). To optimize the results, we do not generate autocomplete suggestions using the full working model as input. Instead, we slice the model according to multiple (potentially overlapping) dimensions and generate suggestions for each slice. This generates a more varied style and a higher number of suggestions and enables the designer to also focus on the types of suggestions she is more interested in (e.g. attribute suggestions vs class suggestions).

The slicing patterns have been thoroughly designed taking into account the information and encoding of the NLP models to take full advantage of them. Each type of slice focuses on a specific type of suggestion. For instance, if we want to generate attribute suggestions, it is better to slice the model isolating the class for which we want to generate the attribute suggestions so that the NLP recommendations are more focused around the semantics of that class and avoid noise coming from other not-so-close classes in the model. There is clearly a trade-off of how much content should be included in each slice depending on the goal. We have refined our current patterns based on our experimental tests.

In short, in each iteration (steps B.1–C.2), we slice the model according to these patterns:

- one slice that contains all the classes in the model after removing their features (attributes and relationships);
- one slice for each class C in the model (keeping its attributes and dangling relationships); and
- one slice for each pair of classes (keeping its attributes and dangling relationships). These slices aim to suggest new classes, attributes and relationships, respectively, as we explain in Section 2.2.

Step B.2. Querying the NLP models and historical data to obtain word suggestions Given a slice, we start by extracting the element names. They become the list of positive words employed to query the two NLP models (i.e., general knowledge and contextual knowledge). The historical data is used to provide negative words when querying the NLP models. Indeed, if the same list of possible words was used in the past to query the NLP models and the designer rejected a suggestion, that suggestion is stored in the historical data (as explained next in Step C.2), and used as a negative case here.

[6] https://nlp.stanford.edu/projects/glove/, https://wikipedia2vec.github.io/wikipedia2vec/pretrained/, https://code.google.com/archive/p/word2vec/

Each query returns a list of new word suggestions sorted by the partial ordering relation (e.g., euclidean distance) between the embeddings of the initial list of words (i.e., the element names extracted from the model slice) and each suggestion. Therefore, the result after querying the two NLP models for each model slice returns two different lists of related concepts, sorted by shorter to longer distance between embeddings (i.e., sorted by relatedness) that we use to prioritize our suggestions. By default, we merge the two lists (the one coming from the contextual knowledge and the one from the general NLP models) into a single sorted list. If a word appears in both lists, the position in which the word appears in the merged list is that whose distance to the slice is smaller (i.e., the relatedness to the slice is higher).

This process can be customized. Our framework is parametrizable in two ways: (i) you can select the number of suggestions to receive at once, and (ii) customize how the the two lists should be prioritized by defining a weight parameter. Regarding the latter, as previously said, by default, our engine mixes the recommendations coming from both sources into only one sorted list. Nevertheless, we provide a parameter to assign different weights to the two sources of knowledge, gn, a value in the range [0..1], where gn=0 means that the user does not want general knowledge suggestions at all, and gn=1 that she only wants general knowledge suggestions. The weight assigned to the contextual knowledge will be $1 - gn$. This prioritization can be used to only get contextual information suggestions, general ones, give different weights to each of them (so that they appear higher in the list) or even to ask for two different lists, which helps trace where the suggestions come from, improving the explainability.

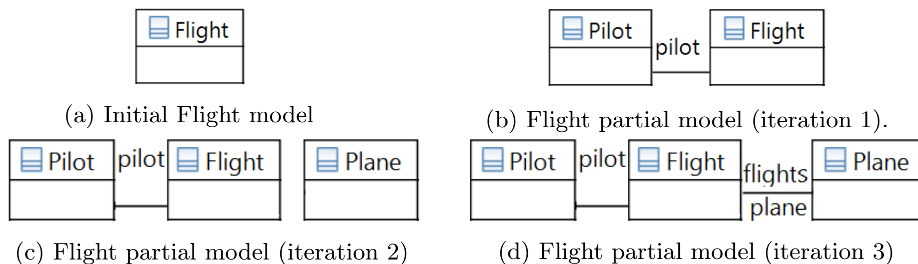
Step B.3. Morphological analysis Before building the potential model elements that will be presented to the designer, we perform some final processing of the lists to remove/refactor some candidate suggestions.

In particular, we use auxiliary NLP libraries [13,7] to perform a morphological analysis of each word (Part-of-Speech (POS) tagging) followed by a lemmatization [7] process, paying especial attention to inflected forms. For instance, if one of the terms returned by a query to an NLP model returns the word *flyers*, our engine lemmatizes it as a verb, resulting in the word *fly*; and as a noun, resulting in the word *flyer*. Therefore, it considers the three words as possible candidates to be the name of a new model element. We also use the POS tag to discard words when they do not apply (for instance, verbs as class names).

Step B.4. Building potential model elements to add As a final step, we transform the refined lists of words into potential new model elements. The interpretation of the right type of model element to suggest depends on the type of slice we are processing.

For slices aiming at new class suggestions, the list of potential concepts refined returned by the NLP morphological analysis refinement step (B.3) is filtered

[7] In linguistics, lemmatization is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma, or dictionary form.



(a) Initial Flight model

(b) Flight partial model (iteration 1).

(c) Flight partial model (iteration 2)

(d) Flight partial model (iteration 3)

Fig. 2: Flight model evolution

to remove verbs, adjectives and plural nouns. After the filtering process, each of the remaining words, w, is a candidate to become a new class named w. For instance, let us assume that we are going to build a model in the domain of flights. Consider that we start from a partial model with a single class named *Flight* and no attributes as Figure 2a shows. After the slicing, querying and lemmatization, we obtain the list of potential concepts refined *[flights, plane, pilots, pilot, flying, fly, airline, airlines, airplane, jet]*. We use the POS tag to filter the list by discarding verbs, adjectives and plural nouns. The list of remaining words is *[plane, pilot, flying, fly, airline, airplane, jet]*. For each word in this list, our algorithm suggests to add a new class with the same name.

For slices aimed at suggesting new features for a class C, for each output word, w, we offer the user three options: (a) add a new attribute named w to C (the user is in charge of selecting the right datatype); (b) add a new class called w and a new relationship between C and w; (c) if there is already a class called w in the partial model, our engine suggests the addition of a relationship between C and w. Continuing with the example, for a slice containing the class *Flight* with no features (Figure 2a), the list of potential concepts refined is: *[flights, plane, pilots, pilot, flying, fly, airline, airlines, airplane, jet]*. For example, when the designer picks the word *pilot*, she will receive the three options above, and she could select, for instance, to add it to the model as a new class and relationship (option b) and obtain the model in Figure 2b.

For slices aimed at discovering new associations, each word, w, is suggested as a new association between the two classes in the slice. For instance, let us assume that we kept building the model and added a new class called *Plane* with no association with the other two (Figure 2c). In this partial model, for the pair of classes *Flight* and *Plane*, our engine suggests the engineer to add associations with names: *flights, pilots, pilot, flying, fly, jet, airplane*. Our designer could select to add two relationships *flights* and *plane* to obtain the model in Figure 2d.

Step B.5. Suggestions provided to user In this step, the generated suggestions are provided to the designer. She can accept, discard or ignore each of them. While the two first options are processed (either by integrating them into the partial model or by marking them as negative test cases), when suggestions are ignored, we do not handle them and they can be presented to the designer in the future again.

2.3 Step C: Update model and historical data

Step C.1. Partial model update In this step, the suggestion(s) accepted by the designer are integrated into the partial model.

Step C.2. Feedback and historical data Every time the designer discards a recommendation, we annotate it as a negative example in order to avoid recommending it again and to guide the NLP model in an opposite direction (i.e., providing the concept as a negative case). Note that the more complete the partial model is and the more feedback we have, the more accurate our suggestions will be.

3 Tool Support

We present the implementation of our architecture in Fig. 1. The source code and pretrained NLP models to reproduce the experiments are available in our Git repository[8].

The text preprocessing algorithm that generates the **domain corpus of text** is implemented as a Java program that reads the input text documents, removes all special characters and merges them into a single textual file. The resulting file only contains words, line breaks and spaces.

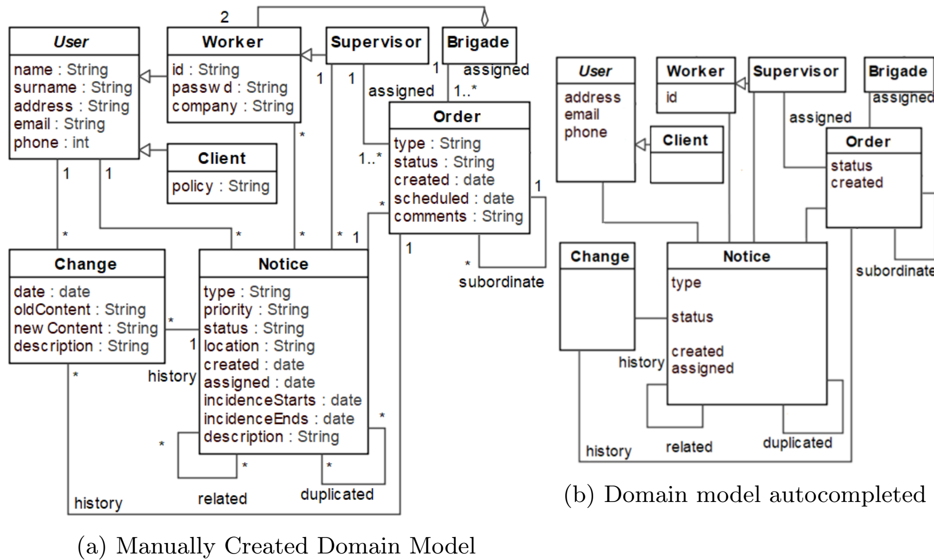
To build the **NLP models** we use GloVe [31], which is an unsupervised learning method that creates word embeddings via an statistical data analysis. It is trained on the entries of a global word-word co-occurrence matrix, which tabulates how frequently words co-occur with one another in a given corpus. Populating this matrix requires a single pass through the entire corpus to collect the statistics, which makes it an efficient method. Note that, while different methods for the computation of word embeddings (e.g., GloVe, word2vec and FastText) differ in its implementation, they can be used equally for the purposes of this work. We have used the Stanford's implementation of GloVe written in Python.

Our NLP component encapsulates two GloVe models, one trained with general data and one with contextual project one (when available). We have created a simple Java library with the necessary methods to create, train, load and query these two NLP models. This library provides functions such as `get_suggestions(nlp_model, positive_concepts, negative_concepts, num_suggestions)`.

As auxiliary **NLP tools** for morphological analysis and lemmatization, we use WordNet [13], which is part of the Python NLTK (Natural Language Toolkit). We query WordNet to obtain the parts of speech of words (i.e., noun, verb, adjective, etc.) and use its lemmatization tool.

Our implementation supports models in EMF (Eclipse Modeling Framework)[9] [37] format. Since this framework is implemented in Java and our engine

[8] `https://github.com/modelia/model-autocompletion`
[9] `https://www.eclipse.org/modeling/emf/`



(a) Manually Created Domain Model

(b) Domain model autocompleted

Fig. 3: Emasa Domain Models

needs to heavily interact with it, the **Model Recommendation Engine** is implemented in Java, too. For example, our engine uses the EMF API to read the input domain model, represented as a UML class model and slices it. The engine is in charge of orchestrating also the previous Python components and implements the suggestion algorithm described in Section 2.

Finally, the **Historical Data** component stores feedback from the designer. This feedback is stored for each user and model, i.e., it keeps track of the suggestions that the designer has discarded for each model. The discarded suggestions are used to both avoid suggesting them again and use them as negative cases from which we also learn. Given the way in which GloVe word embeddings are encoded, it enables the search of words that are both as close as possible to a set of words (positive cases) and as far as possible to other set of words (negative cases). The recommendation engine uses this feature when querying the NLP models.

4 Validation

4.1 Case study and experiment setup

Let us consider an example of an industrial project: the introduction of a notice management system for incidents in the municipal water supply and sewage in the city of Malaga, Spain. The Malaga city hall and the municipal water and sewage company (EMASA) started a project to manage the incidents that clients and citizens notify to have occurred either in private properties or public locations. This project replaced the previous process that was handled via phone calls and paper forms. In this project, contextual knowledge can be derived from the project documentation (e.g. requirements specification). Meanwhile, general

knowledge can be extracted from texts in Wikipedia entries, Google News, or similar sources covering general water supply and sewage issues. The project developers produced manually the domain model of this system shown in Fig. 3a. The goal of this section is to evaluate how well our approach can regenerate this manual model by means of autocompleting partially seeded models to show the quality and benefits of our proposal.

For this case study, the contextual model was trained with the project documentation provided by the client: slides (21), forms and the software requirement specification document (78 PDF pages) that after being preprocessed turned into a 48Kb text file with 7.675 words[10]. For the general knowledge model, we have reused the pretrained word embeddings available at `https://nlp.stanford.edu/projects/glove/`, which has been trained with the corpus of text from Wikipedia.

As a preliminary evaluation, we have split the domain model from our case study into 5 different models, each simulating a potential partial model with a single class and no attributes/relationships. Fig. 4 shows our five initial models. The goal is to reconstruct the model shown in Fig. 3a from each partial model. We have parameterized our engine to provide 20 suggestions per round and opted to receive contextual and general knowledge separately.

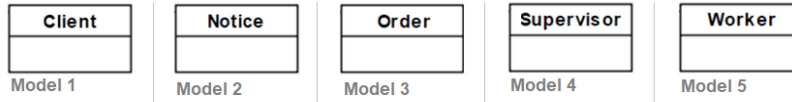


Fig. 4: Initial models

We have automated the reconstruction process by automatically simulating the behaviour of a designer using our framework. As we know the final target (the full model) we can automatically accept/reject the suggestions based on whether they do appear in the full model or not. Accepted ones are integrated in the (now extended) partial model. New rounds of suggestions are requested until no more acceptable suggestions are received. Note that this evaluation can be regarded as a worst case scenario as the evaluation criteria is very strict: in a real-case scenario, a designer could consider as good suggestions a broader set of scenarios as there is no single and unique correct model for any domain. And, obviously, real designers can completely stop the suggestions at any time, edit the model manually and then resume the suggestions again.

As part of this evaluation, we consider the answer to the following research questions for our case study:

- **RQ1.** Recall: what percentage of the elements of the final model are we able to reconstruct?

[10] These documents are not publicly available due to industrial property right. Nevertheless, the software artifacts derived from them are available in our Git repository.

- **RQ2.** Precision: what percentage of suggestions are accepted and integrated into the domain model?
- **RQ3.** Source of accepted suggestions: what percentage of accepted suggestions are coming from general knowledge and contextual knowledge?
- **RQ4.** Performance: how does our prototypical implementation perform?

4.2 Recall (RQ1)

Our experiments show that, in all cases, our simulation has been able to reconstruct all classes; for the attributes it has identified an average of 9.67 out of the 27 that the complete model has with a standard deviation of 0.58 (i.e. 9.67 ± 0.58), 9.67 ± 0.58 out of the 13 relationships; and 6.67 ± 0.6 out of the 7 association names. In total, it has identified an average of 34 ± 1.73 out of the 55 model elements, which is approximately a 62% of the model. As an example, Fig. 3b shows the autocompletion produced starting from an empty class *Notice*.

4.3 Precision (RQ2)

On average, our framework has been queried 15.67 times (each query returning 20 possible suggestions for each source of knowledge), with a standard deviation of 5.69 (15.7 ± 5.69). This means that our designer bot has received, on average, a total of 626.7 ± 227.4 suggestions. It has accepted an average of 25.67 ± 0.58 suggestions, resulting on 34 ± 1.73 model elements added to the domain model. Thus, the precision of our approach is 4.46%. Although in absolute terms it seems low, note that the average number of suggestions accepted per set of suggestions is 1.79 ± 0.62 suggestions and that our partial model includes very limited knowledge (a single class name). Furthermore, our automatic acceptance criteria is very strict, i.e., it considers a single target domain model as the ground truth. In reality, several alternatives models are feasible so a human designer might have accepted suggestions rejected by the bot.

4.4 Source of accepted suggestions (RQ3)

On average, an 85.7% of accepted suggestions came from the contextual knowledge. This is expected as this is a very particular domain for which it is difficult to assume there is a rich-enough description in a general knowledge source. Nevertheless, the general knowledge has complemented the contextual one and has helped discovering implicit knowledge in the contextual descriptions.

4.5 Performance (RQ4)

We have measured independently the execution time that each component of our framework and its main steps takes. The experiments have been executed in a machine with Windows 10, an Intel i7 8th generation processor at 1.80 GHz, 16Gb of RAM memory and 4 cores with 8 logical processors.

We have observed that, for our case study, all the times are under one second. For instance, the model slicing takes on average 19 milliseconds (ms), the build

of the potential model suggestions takes on average 1 ms, etc. The only step that heavily affects the overall performance of our framework is the querying of the NLP models. Thus, we have paid special attention to that.

On the one hand, the training of the contextual knowledge model using the project documents only took several ms. It resulted in a file that contains the word embeddings with a size of 121 KB. After training, the time to load the word embeddings—this is done only when the system is initialized—as well as the time to query the model are negligible. This is due to the small size of the contextual data (text documents, the derived word embeddings).

On the other hand, for the general knowledge model, the file with the pre-trained embeddings has a size of 989 MB and it takes around 32 seconds to be loaded. Once loaded, a query takes several seconds. For this reason, we plan in the future to replace the Python implementation of GloVe with a pure C implementation that will improve the performance considerably.

5 Related Work

Our work is related to works on autocompletion in software development, extraction of models from text and modeling assistants. In the following we give an overview of the state of the art in each group and discuss the differences with our own proposal.

Autocompletion in software development. Development tools can offer different types of recommendations to software developers [33,16,19]. Among them, *code completion* [25,5] is a standard feature of IDEs. A similar notion is *query (auto)completion* in information retrieval [36], *e.g.*, search engines. Both approaches propose textual completions and use a combination of frequent patterns, information about the context and historic data to provide useful suggestions.

In this paper, we target model completion rather than source code or query completion. While some of the techniques employed in these problems are related, there are fundamental differences among them:

- Code and query completion place a very strong emphasis the analysis of historic data. This requires a large repository of examples, which is usually not available at such a large scale in the case of modeling. For this reason, similarity and relatedness, which play a complementary role in code and query completion, are the key components of model completion.
- Some coding activities are predictable and repetitive (e.g., define a constructor to initialize all attributes of an object) so code completion can provide useful suggestions simply by considering frequent patterns. On the other hand, models tend to be one-of-a-kind: even when considering a ubiquitous domain (e.g., the structure of an organization), the vocabulary, constraints and level of detail may vary from one model to another.
- Code and query completion is typically *local*: completions are proposed for the current method or query. Meanwhile, model completion can be local or *global*, i.e., identify missing elements in the entire model.

- In addition to proposing relevant missing elements, model completion needs to assign a category/type to these proposed elements (attribute, class, relationship) and establish how it relates to the existing model, e.g. a relationship between classes X and Y.

Model extraction from textual requirements. Several approaches aim to generate software models from textual specifications. Among them, some works extract structural information such as UML class diagrams [22,2,34,20], entity-relationship diagrams [17] or domain ontologies [6,24,9,38]. Others focus on other type of information, like variability [3,32] (commonalities and differences among the products in a software product line) or behavior [14] (such as the workflow in a business process). Their goal is not the completion of a partial model, but the construction of a new model from scratch.

Even though the type of models varies, all these approaches rely on Natural-Language Processing (NLP) techniques and tools and share similar subtasks as ours. Nevertheless, they do not take into account the partial model, as we do in the context of this paper. This means that their approach cannot be guided by the designer nor can they integrate any type of feedback during the model creation process. As a consequence, their predictions will be less accurate.

Modeling assistants. Several tools apply model autocompletion with different goals. For instance, [23] analyzes designer actions in the GUI of a model editor to detect ongoing high-level activities from a predefined catalog (*e.g.*, a refactoring) and propose actions to auto-complete the activity. [21] suggests meaningful names for methods and UML model elements. Meanwhile, [35] suggests completions of a domain-specific model in order to satisfy well-formedness rules. These completions are proposed by used either a relational model finder (Alloy) or a constraint solver. Another approach, [11], clusters classes in a metamodel repository according to a similarity metric to identify related classes. Then, it recommends related classes to those in the partial model. None of them leverage any project textual documents to improve the recommendations. Moreover, two commercial software development tools provide AI-powered assistants: ServiceStudio from OutSystems [30] and Mendix Assist [26], based on existing models in their private repositories. As before, they do not use any type of project document as additional input. Finally, [1] recommends related models based on knowledge from Google Books, but it does not consider feedback nor contextual knowledge as we do.

6 Conclusions

This exploratory paper has proposed a model recommendation engine that, once fed with textual descriptions of a domain, generates autocompletions for domain models under development. This is a first step towards a more general modeling assistant that effectively helps modelers specify better models faster.

As further work, we plan to integrate in our framework other types of information (such as past models created by the same company in the same or similar domains and general ontologies, e.g., SUMO) to provide richer suggestions. This will imply dealing with prioritization/inconsistencies among the different sources.

Usability will be a key point to ensure the framework is well accepted by software designers. We will explore the optimal parameters for our system such as the number of suggestions, the confidence threshold to suggest a new model element, the timeliness (i.e., when to trigger the suggestions) and the level of automation (i.e., they are automatically sent to the user versus they are provided only on-demand). We will ensure that our approach can be effectively used by carrying out an empirical evaluation with a group of experienced designers.

We will keep refining the techniques presented in this paper, e.g., accounting for aggregation, composition, and generalization relations and suggesting also data types and potentially missing constraints beyond just new elements. We will also like to extend the type of suggestions we offer to include the replacement and removal of elements. Finally, we will study the application of our approach on other types of models and modeling languages and the exploitation of other types of NLP models in software modeling.

References

1. Agt-Rickauer, H., Kutsche, R., Sack, H.: Automated recommendation of related model elements for domain models. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proc. of MODELSWARD'18. vol. 991, pp. 134–158 (2018)
2. Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F.: Extracting domain models from natural-language requirements: approach and industrial evaluation. In: Baudry, B., Combemale, B. (eds.) Proc. of MODELS'16. pp. 250–260 (2016)
3. Bakar, N.H., Kasirun, Z.M., Salleh, N.: Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. J. Syst. Softw. **106**(C), 132–149 (Aug 2015)
4. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., et al.: Language models are few-shot learners (2020), https://arxiv.org/abs/2005.14165
5. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: ESEC-FSE'09. pp. 213–222 (2009)
6. Buitelaar, P., Cimiano, P., Magnini, B.: Ontology learning from text: methods, evaluation and applications, vol. 123. IOS press (2005)
7. CEA NLP tech: LIMA: LIbre Multilingual Analyzer. https://github.com/aymara/lima/wiki/DeepLima-beta#the-lima-multilingual-nlp-tool (2020)
8. Conesa, J., Olivé, A.: A method for pruning ontologies in the development of conceptual schemas of information systems. In: JoDS V. pp. 64–90 (2006)
9. Dahab, M.Y., Hassan, H.A., Rafea, A.: Textontoex: Automatic ontology construction from natural english text. Expert Systems with Applications **34**(2), 1474–1480 (2008)
10. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding (2018), http://arxiv.org/abs/1810.04805
11. Elkamel, A., Gzara, M., Ben-Abdallah, H.: An UML class recommender system for software design. In: Proc. of AICCSA'16. pp. 1–8 (2016)
12. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
13. Fellbaum, C.: WordNet: An Electronic Lexical Database. Bradford Books (1998), https://wordnet.princeton.edu/

14. Friedrich, F., Mendling, J., Puhlmann, F.: Process model generation from natural language text. In: Proc. of CAISE'11. pp. 482–496 (2011)
15. Ganser, A., Lichter, H.: Engineering model recommender foundations. In: Proc. of MODELSWARD'13. vol. 19, pp. 135–142 (2013)
16. Gasparic, M., Janes, A.: What recommendation systems for software engineering recommend. J. Syst. Softw. **113**(C), 101–113 (2016)
17. Gomez, F., Segami, C., Delaune, C.: A system for the semiautomatic generation of er models from natural language specifications. Data & Knowledge Eng. **29**(1), 57–81 (1999)
18. Grave, E., Bojanowski, P., Gupta, P., Joulin, A., Mikolov, T.: Learning word vectors for 157 languages. In: Proc. of the International Conference on Language Resources and Evaluation (LREC'18) (2018)
19. Harel, D., Katz, G., Marelly, R., Marron, A.: Wise computing: Toward endowing system development with proactive wisdom. Computer **51**(2), 14–26 (2018)
20. Harmain, H.M., Gaizauskas, R.J.: Cm-builder: A natural language-based case tool for object-oriented analysis. Automated Software Engineering **10**, 157–181 (2003)
21. Kuhn, A.: On recommending meaningful names in source and UML. In: Proc. of RSSE'10. pp. 50–51 (2010)
22. Kumar, D.D., Sanyal, R.: Static UML model generator from analysis of requirements (SUGAR). In: Proc. of ASEA'08. pp. 77–84 (2008)
23. Kuschke, T., Mäder, P.: Pattern-based auto-completion of UML modeling activities. In: Proc. of ASE'14. ASE'14 (2014)
24. Lee, C.S., Kao, Y.F., Kuo, Y.H., Wang, M.H.: Automated ontology construction for unstructured text documents. Data & Knowledge Eng. **60**(3), 547–566 (2007)
25. Marasoiu, M., Church, L., Blackwell, A.F.: An empirical investigation of code completion usage by professional software developers. In: Proc. of PPIG'15. p. 14 (2015)
26. Mendix: Mendix assist (2020), `https://www.mendix.com/platform/\#assist`
27. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Proc. of NIPS'13. vol. 2 (2013)
28. Mussbacher, G., Combemale, B., Kienzle, J., et al.: Opportunities in intelligent modeling assistance. Software and Systems Modeling (2020)
29. Olivé, A.: Conceptual modeling of information systems. Springer (2007)
30. OutSystems: (2020), `https://www.outsystems.com/p/low-code-platform/`
31. Pennington, J., Socher, R., Manning, C.D.: GloVe: Global vectors for word representation. In: Proc. of EMNLP'14. pp. 1532–1543 (2014)
32. Reinhartz-Berger, I., Kemelman, M.: Extracting core requirements for software product lines. Requirements Engineering **25**(1), 47–65 (2020)
33. Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE software **27**(4), 80–86 (2009)
34. Sagar, V.B.R.V., Abirami, S.: Conceptual modeling of natural language functional requirements. Journal of Systems and Software **88**, 25 – 41 (2014)
35. Sen, S., Baudry, B., Vangheluwe, H.: Towards domain-specific model editors with automatic model completion. Simulation **86**(2), 109–126 (2010)
36. Shao, T., Chen, H., Chen, W.: Query auto-completion based on word2vec semantic similarity. Journal of Physics: Conference Series **1004**(1), 12–18 (2018)
37. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
38. Wong, W., Liu, W., Bennamoun, M.: Ontology learning from text: A look back and into the future. ACM Computing Surveys (CSUR) **44**(4), 1–36 (2012)