Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00942-6

THEME SECTION PAPER

Automated, interactive, and traceable domain modelling empowered by artificial intelligence

Rijul Saini[1] · Gunter Mussbacher[1] · Jin L. C. Guo[2] · Jörg Kienzle[2]

Received: 6 April 2021 / Revised: 19 August 2021 / Accepted: 13 October 2021

Abstract
Model-Based Software Engineering provides various modelling formalisms for capturing the structural, behavioral, configuration, and intentional aspects of software systems. One of the most widely used kinds of models—domain models—are used during requirements analysis or the early stages of design to capture the domain concepts and relationships in the form of class diagrams. Modellers perform domain modelling to transform the problem descriptions that express informal requirements in natural language to domain models, which are more concise and analyzable. However, this manual practice of domain modelling is laborious and time-consuming. Existing approaches, which aim to assist modellers by automating or semi-automating the construction of domain models from problem descriptions, fail to address three non-trivial aspects of automated domain modelling. First, automatically extracted domain models from existing approaches are not accurate enough to be used directly or with minor modifications for software development or teaching purposes. Second, existing approaches do not support modeller-system interactions beyond providing recommendations. Finally, existing approaches do not facilitate the modellers to learn the rationale behind the modelling decisions taken by an extractor system. Therefore, in this paper, we extend our previous work to facilitate bot-modeller interactions. We propose an algorithm to discover alternative configurations during bot-modeller interactions. Our bot uses this algorithm to find alternative configurations and then present these configurations in the form of suggestions to modellers. Our bot then updates the domain model in response to the acceptance of these suggestions by a modeller. Furthermore, we evaluate the bot for its effectiveness and performance for the test problem descriptions. Our bot achieves median F1 scores of 86%, 91%, and 90% in the *Found Configurations*, *Offered Suggestions*, and *Updated Domain Models* categories, respectively. We also show that the median time taken by our bot to find alternative configurations is 55.5ms for the problem descriptions which are similar to the test problem descriptions in terms of model size and complexity. Finally, we conduct a pilot user study to assess the benefits and limitations of our bot and present the lessons learned from our study in preparation for a large-scale user study.

Keywords Domain model · Natural language (NL) · Natural language processing (NLP) · Machine learning (ML) · Neural networks · Descriptive model · Predictive model · Bot–Modeller interactions · Traceability information model · Traceability knowledge graph

1 Introduction

Model-Driven Software Engineering advocates the use of various modelling formalisms such as goal models, feature models, class diagrams, state machines, and workflows for supporting software development. These formalisms help in capturing the intentional, configuration, structural, and

Communicated by L. Burgueño, J. Cabot, M. Wimmer & S. Zschaler.

✉ Rijul Saini
rijul.saini@mail.mcgill.ca

Gunter Mussbacher
gunter.mussbacher@mcgill.ca

Jin L. C. Guo
jguo@cs.mcgill.ca

Jörg Kienzle
joerg.kienzle@cs.mcgill.ca

[1] Department of Electrical and Computer Engineering, McGill University, Montréal, Canada

[2] School of Computer Science, McGill University, Montréal, Canada

behavioral aspects of software systems. One such formalism is class diagrams, which are used to represent high-level domain models during requirements analysis or the early stages of design. These models capture the structural aspects of the problem domain in the form of classes, attributes, relationships, and association cardinalities. The practice of transforming domain problem descriptions written in natural language (NL) into concise and analyzable domain models is known as domain modelling. Since domain modelling is time-consuming and requires modelling skills and experience, several works already go in the direction of extracting domain models either semi-automatically or automatically [2,19,20]. However, there are certain challenges which remain in this direction. First, the domain models extracted by existing approaches are not accurate enough to be used in software development or for teaching purposes directly or with minor modifications. Second, no existing approach facilitates bot-modeller interactions to include those modelling decisions which are preferred by a modeller over the current decisions. Finally, no existing approach enables modellers to gain insights into the domain modelling decisions taken by an extractor system.

In this paper, we extend the architecture proposed in our previous work [40] to facilitate automated, interactive, and traceable domain modelling. We address the following three research questions:

RQ1 (Effectiveness): How effective are the bot responses in providing suggestions and updating domain models? In this paper, we evaluate the effectiveness of our bot in finding alternative configurations, presenting these configurations to modellers in the form of suggestions, and updating the extracted domain models. We find that our bot achieves median F1 scores of 86%, 91%, and 90% in the *Found Configurations*, *Offered Suggestions*, and *Updated Domain Models* categories, respectively.

RQ2 (Performance): Does our tool discover alternative configurations and generate suggestions in practical time? We evaluate the time performance of our bot by measuring the time taken by our bot for the test problem descriptions in finding alternative configurations, generating corresponding recommendations in NL, and updating the domain models with the changes accepted by a modeller. We find that our bot finds alternative configurations, suggests these configurations to modellers in NL, and updates an extracted domain model in practical time, i.e., median time of 55.5ms, 11ms, and 19ms, respectively, for the problem descriptions which are similar to the test problem descriptions in terms of size and complexity.

RQ3 (Usability): What are the benefits and limitations of our tool in supporting bot-modeller interactions? The usefulness of our proposed tool ultimately depends on whether modellers find the tool helpful in real settings. Therefore, we conduct a pilot user study and present the lessons learned in preparation of a large-scale user study.

This paper extends our earlier work [36–40] which includes:

1. Architecture design of a domain modelling bot called DoMoBOT [40]. The architecture is based on the ModBud framework [39] for extracting domain models from problem descriptions in NL, facilitating modelling decisions traceability, and supporting bot-modeller interaction.
2. Foundation of bot-modeller interactions [40].
3. A Traceability Information Model (TIM) to enable traceability of modelling decisions in forward and backward directions [38].
4. Evaluation results of our proposed approach for extracting domain models [36,37,40] as well as for enabling modelling decisions traceability [38].

We extend this work by adding the *Recommendation Component* and the *Query Answering Component* to the architecture proposed in our previous work. We propose an algorithm to discover alternative configurations during bot-modeller interaction. These configurations are presented by the bot in the form of suggestions to modellers in NL. Our bot then updates the extracted domain model in response to the acceptance of these offered suggestions by a modeller. In this paper, we also present a taxonomy of potential modeller actions as well as a classification strategy for modeller intents. These actions can be performed by a modeller during bot-modeller interactions and these intents can be triggered by the bot or modeller.

The remainder of this paper is organized as follows. Section 2 provides background information and Sect. 3 surveys related work. Section 4 describes the extended architecture of DoMoBOT. Section 5 presents our experiment design and research questions. Section 6 presents and discusses the results. Section 7 describes the threats to the validity of our study. Finally, Sect. 8 draws conclusions and highlights future work.

2 Background

In this section, we provide an overview of the natural language processing (NLP) and machine learning (ML) techniques used in this paper for extracting the domain model elements from problem descriptions expressed in NL. We further explain the concepts we use to enable bot-modeller interactions and traceability of domain modelling decisions.

2.1 Use of NLP and ML techniques

We use rule-based NLP for extracting the domain models from problem descriptions written in NL. Rule-based NLP represents some rules motivated by the state-of-the-art model extraction rules used in related work [2,35]. In rule-based NLP, heuristics based on linguistic features, e.g., part-of-speech (POS) tags and syntactic dependency labels are used. POS tags indicate the part of speech and also grammatical categories (tense, case etc.) in some cases. For example, a POS tag *NNS* indicates a plural noun and a POS tag *VBD* represents a verb in past tense. Also, the dependency labels represent the syntactical structure of a sentence with a set of directed binary grammatical relations that hold among the word tokens of this sentence. In rule-based NLP, these linguistic features are used to find the subject, object, and their associated relationships in addition to the noun entities which may represent domain concepts.

On the other side, there are certain scenarios where extraction of concepts and their relationships for a given problem description goes beyond what is possible with rule-based NLP. First, categorization of a domain concept into a *class* or an *attribute* using extraction rules is not always possible. For example, the "position" concept can be modelled as a *class* ("An employee can be hired for multiple positions.") or as an attribute ("The system records the position of employees.") depending upon the scenario in a problem description. Second, the information about attribute types is often not provided in the problem description and also some attributes are used with specific types, e.g., the "weight" concept is often modelled as an *attribute* with *float* type. Third, extracting relationships or modelling patterns which are hidden in the text of a problem description is not possible without obtaining context information of the participating concepts. For example, the sentence "Exams can be of types online and classroom." can be modelled by creating the "KindOfExam" enumeration type or by creating a generalization relationship between the superclass (Exam) and its subclasses (Online and Classroom). However, another sentence in the same problem description "Online exams have unique IDs. Classroom exams are identified by room number.", indicates that concepts (Online exams and classroom exams) have different attributes and then it becomes essential to model this scenario using generalization only. Therefore, obtaining context information and external knowledge of domain concepts is imperative in these cases.

To address the above scenarios, we use supervised ML classifiers, e.g., Linear Discriminant Analysis[1] and Logistic Regression[2] models for predicting the concepts category (class or attribute) and types (*class* for *class* concept or *string*, *float*, *enumeration*, *date*, *time*, or *integer* for *attribute* concept) [37]. In addition, we use neural networks for predicting the relationship type (e.g., association or composition) or pattern (Player-Role) [36] as well as for predicting the cardinalities (e.g., zero-to-one) [40]. In these classifiers, we use pre-trained word embeddings to transform the words into real-valued vectors. In word embeddings, vectors of similar words have a similar encoding and lie closer to each other in the semantic space representation. For example, vectors of words "weight" and "height" will be closer to each other than the vector of "university". For learning the context information, we use Bidirectional Long Short-Term Memory (BiLSTM) neural networks.[3] LSTM [16] neural networks are designed to make use of sequential information, e.g., sentences and they are capable of handling long-term dependencies. BiLSTM is comprised of two LSTMs to process a sentence in forward direction and backward direction. The results from both directions are then merged by the BiLSTM model. Finally, we evaluate the accuracy of our extracted domain models using problem descriptions which were never seen or touched during the development activities related to the model extraction process. The average precision and recall scores of extracted models using our proposed BOT are 90% and 77%, respectively [38].

2.2 Bot-Modeller interactions and traceability

To enable bot-modeller interactions and traceability of domain modelling decisions, we further process the results of the domain model extraction process using our proposed algorithm and a TIM [38], respectively. The model extraction process can result in multiple solutions (with different probability scores) for the same concept. For example, ML models predict association as well as composition relationships between *Team* and *Player* classes with different probability scores. On the other side, we use a TIM for automated traceability of domain modelling decisions so that modellers can gain insights into the decisions taken by an extractor system [38]. A TIM is composed of two building blocks—trace artifacts and trace links. A trace artifact represents the unit of data and a trace link characterizes trace artifacts with the same or similar structure (syntax) and/or purpose (semantics) [15]. There are several advantages of a TIM such as automated traceability handling, validation, and analyses [24,25,32]. Furthermore, during instantiation of our tool, we use a *Traceability Knowledge Graph* to persist these trace artifacts and trace links. The *Traceability Knowledge Graph* stores the traceability information of extracted model elements in the form of nodes (entities which can hold any number of attributes), relationships (semantically relevant

1 https://en.wikipedia.org/wiki/Linear_discriminant_analysis.

2 https://en.wikipedia.org/wiki/Logistic_regression.

3 https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks.

links between two nodes), and properties (can be assigned to both nodes and relationships).

Finally, we evaluate the traceability of domain modelling decisions [38] where our bot achieves an overall median F2 Score of 82.04% in facilitating traceability of domain model elements. In this paper, we extend the architecture of our proposed bot to facilitate bot-modeller interactions and evaluate the bot's responses during these interactions.

3 Related work

In this section, we situate our work with respect to the use of NLP and ML techniques in extracting or building models, traceability models, and bots or interactive interfaces for modelling.

3.1 Use of NLP or ML techniques

Several works exist that use NLP-based techniques alone to extract UML diagrams or domain models from text [19,20, 22,28,41,47]. Also, the approach by Arora et al. [2] uses model extraction rules with complementary rules from the information retrieval literature to extract a domain model. In contrast, our proposed bot combines NLP and ML techniques to extract domain models with higher accuracy. Robeer et al. propose an approach which requires user stories in a particular format for automatic extraction of conceptual models from them [35]. However, our proposed bot aims to process problem descriptions written in free-form text. Furthermore, several approaches exist which use ML techniques for model construction or model transformation. For example, Bencomo et al. demonstrate how to define a requirements-aware runtime model for the given requirements specification using partially observable Markov decision processes [4]. Also, Burgueño et al. propose an approach to automatically infer model transformations from sets of input-output model pairs using LSTM neural networks [7]. In our previous work [36,40], we use BiLSTM to learn the context information of domain concepts and predict their associated relationships or modelling patterns. In this paper, we propose an algorithm which discovers alternative configurations based on the results generated after combining NLP and ML techniques during model extraction.

3.2 Use of traceability models

Several works exist which provide traceability between different artifacts, e.g., traceability between UML diagrams and target models [48], requirements-level and code aspects [42], class entities and sections in NL documents [9], and tracing information when concerns are reused [46]. In addition, Hübner et al. develop an interaction log-based trace link creation approach to continuously provide trace links during a project [18] and Houmb et al. present a security requirements elicitation and tracing approach [17]. In contrast, our approach provides traceability between NL problem descriptions and the generated domain models. Moreover, works exist which define and use traceability metamodels, e.g., trace metamodel for Domain-Specific Modelling Languages [5], trace metamodel to capture non-functional requirements and their relations [21], and specification of a metamodel in the context of software product line [1]. Also, Cleland-Huang et al. provide a four-layered model for defining traceability metagraphs, and then introduces a new technique for modelling reusable traceability queries [12]. On the other side, in our work, we present a *TIM* for automated traceability of domain modelling decisions. Schlutter and Vogelsang propose an approach to transform a set of heterogeneous NL requirements into a knowledge representation graph [45]. In contrast, we use a *Traceability Knowledge Graph* that stores the traceability information of extracted model elements. In the literature, some works also propose traceability benchmarking and discuss its benefits [3,10,11] such as support for evaluation and comparison of traceability techniques. To the best of our knowledge, no existing approach provides end-to-end traceability for domain modelling decisions.

3.3 Use of bots or interactive interfaces

Work exists which proposes recommendation systems. These recommendation systems offer suggestions to the modellers based on certain factors, e.g., similarity between the current content (models) and the existing models in knowledge repositories [6,8,13,23,29]. In addition, a research plan is proposed by Savary-Leblanc et al. [43] which aims to create modelling assistants using Artificial Intelligence (AI) techniques for providing suggestions. In contrast, the objectives of our proposed bot are to extract complete domain models with high accuracy from problem descriptions written in NL and to enable modellers to update a part of the extracted model so that in response the bot can update other parts of the extracted model. Furthermore, our bot aims to provide insights into modelling decisions, which goes beyond providing suggestions. Pérez-Soler et al. [34][33] develop a modelling bot and integrate it with social media platforms like Telegram. Their bot uses NLP techniques to interpret the users' inputs in their NL and incrementally builds a metamodel. However, their bot neither generates a complete domain model for a domain problem description nor provides insights into the modelling decisions taken by the model extractor. Mussbacher et al. present a conceptual framework to build intelligent modelling assistants more systematically [30]. On other side, our work focuses on building a modelling assistant for one formalism—domain modelling, and provides a proof-of-concept in the form of a tool.

4 DoMoBOT architecture

In this section, we describe different components which are required to instantiate a domain modelling bot for automatic, interactive, and traceable domain modelling. Figure 1 provides an overview of the architecture, which represents various components to transform the problem description written in NL into a form suitable for model extraction (Component A), to extract domain models automatically using NLP and ML techniques (Components B, C, and D), and to allow modellers to interact with the bot to update the problem description or the extracted domain models and to trace domain modelling decisions (Component E, F, G, and H). In this paper, we add components F and H to the architecture proposed in our previous work [36–40] for facilitating bot-modeller interactions and traceability of modelling decisions. To explain our proposed architecture, we use a small *Problem Description* which represents a *Bank Management System* (PD_{BMS}): *Employees work at different branches and they are identified by an employee number. Employees can be temporary employees or permanent employees. Supervisors are assigned to temporary employees. A customer can open multiple accounts at a branch. An account can be of types - checking, savings, and credit card.*

4.1 Pre-processing component (A)

In this component, we use the spaCy library[4] to process the domain problem description across various steps. First, we use the CoReference Resolution technique (coreferring noun phrases) to identify all noun phrases that refer to the same entity or concept in a problem description in the *CoReference Resolution* step. For example, the "They" word entity refers to the entity "employees" in the first sentence of PD_{BMS}. The output from this step is a *References Dictionary* which represents a map of entities and their corresponding noun phrases. Second, we split the problem description into individual *Sentences* using our custom sentence segmenter in the *Sentence Splitting* step. The custom sentence segmenter uses heuristic rules which are based on dependency labels and part-of-speech (POS) tags of words. The POS tags represent the part of speech or grammatical categories such as tense and the dependency labels indicate the dependencies among words in a sentence to reflect their grammatical relations. The word dependencies and POS tags for the whole text of the problem description are obtained by the spaCy library on the fly during the sentence segmentation process. The custom sentence segmenter not only identifies the boundaries for individual sentences across the text but also the boundaries within a sentence so that a long sentence comprised of multiple *subjects* can be further split into shorter sentences. For example, the first sentence can be split into two - "employees work at different branches" and "employees are identified by an employee number" after obtaining the reference between "they" and "employees" from the *References Dictionary*.

Finally, in the *Linguistic Features Assignment* step, the *Sentences* are transformed into a *Doc Object* where each word is represented as a token. In addition, we use a *Tagger* and *Parser* from the spaCy library to assign POS tags and dependency labels, respectively, to these tokens. Figure 2 illustrates the assignment of POS tags and dependency labels to the fourth sentence of PD_{BMS}. Tables 1 and 2 reflect the meaning of assigned POS tags and dependency labels, respectively. Even though NLP features, e.g., POS tags and dependency labels, are obtained by the spaCy's pre-trained machine learning models in the background, we consider this component as driven by NLP techniques dominantly due to the direct usage of NLP features in this component as shown in Fig. 1.

4.2 Descriptive component (B)

We call this component descriptive because it abstracts a given *Domain Problem Description* and captures only the relevant domain concepts in the form of classes, attributes, relationships, and cardinalities. These domain concepts are derived from the information that is already available in the form of textual problem descriptions. The output from the *Pre-processing Component* in the form of *Processed Doc Object* holds the meta-data for each token of the problem description. Therefore, the *Processed Doc Object* representing the problem description can now be processed with rule-based NLP for extracting concepts, relationships, and cardinalities in this component. First, in the *Concepts Discovery* step, noun chunks are obtained using the spaCy parser which iterates over the *Doc Object* and extracts noun phrases. Next, each noun phrase composed of either a single noun entity representing one concept or conjugated entities which are together considered as one concept, is processed further with rule-based NLP to identify candidate concepts. For example, in the sentence, "Supervisors are assigned to temporary employees", two noun phrases are identified by the parser—"supervisors" and "temporary employees". These two phrases are processed with rule-based NLP to form two concepts—"Supervisor" and "TemporaryEmployee". While forming a concept, the spaCy's lemmatizer is used which performs morphological analysis of words to remove inflectional endings and to return the base or dictionary form of a word (lemma). For example, the lemma form for the "supervisors" token is "Supervisor". The output from this step is a *Concepts Dictionary* which represents a map of tokens along with their corresponding features such as POS tags, dependency labels, lemma form, and position indices in a sentence.

[4] https://spacy.io/.

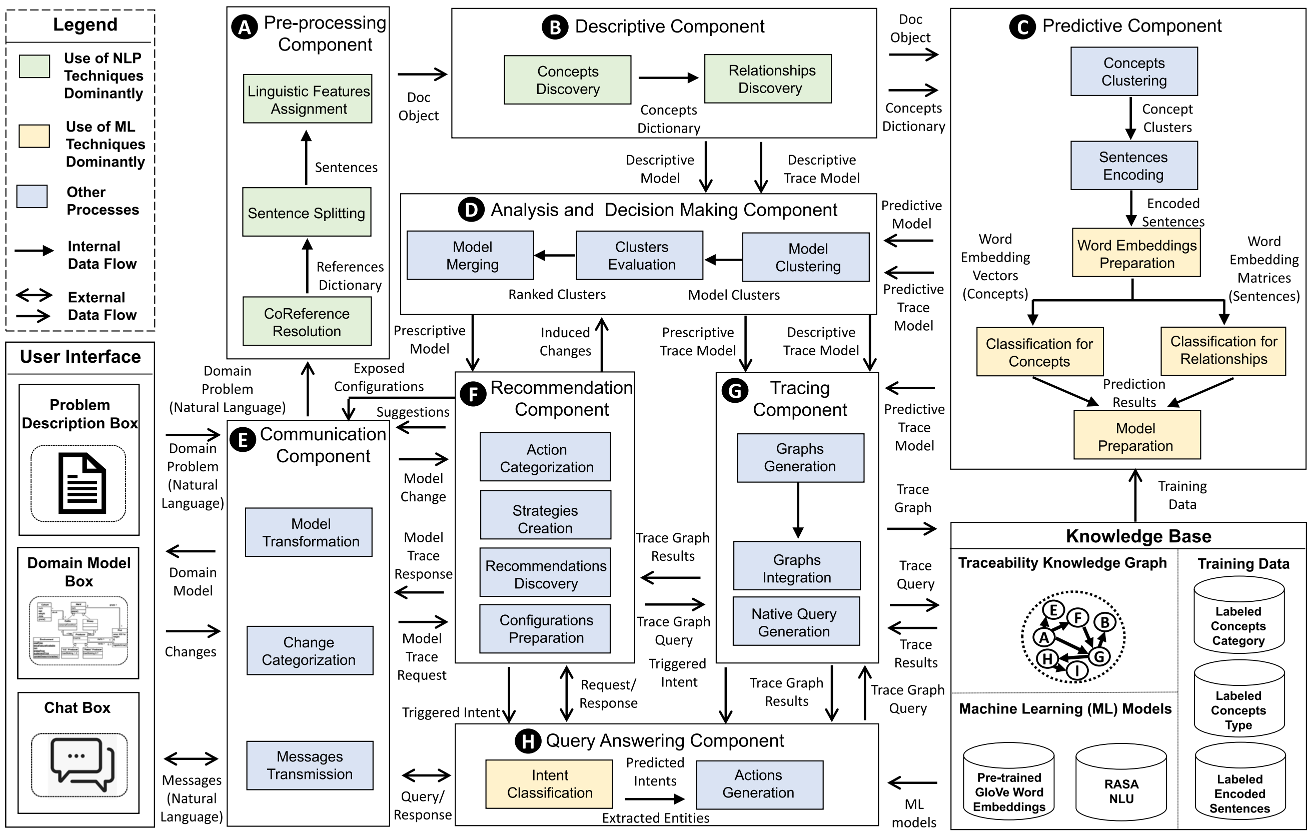


Fig. 1 Proposed Architecture for DoMoBOT

Table 1 Example POS Tags

Tag	Description
DT	Determiner
NN	Noun, singular or mass
MD	Modal
VB	Verb, base form
JJ	Adjective
NNS	Noun, plural
IN	Preposition

Table 2 Example dependency labels

Label	Description
det	Determiner
nsubj	Nominal subject
aux	Auxiliary
dobj	Direct object
amod	Adjectival modifier
prep	Prepositional modifier
pobj	Object of a preposition

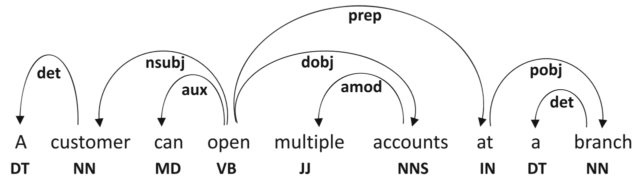

prep
det
nsubj
aux
dobj
amod
pobj
det
A customer can open multiple accounts at a branch
DT NN MD VB JJ NNS IN DT NN

Fig. 2 Dependency Graph for an Example Sentence

Second, the *Concepts Dictionary* from the previous step is used to find the associated relationships of the concepts. In the candidate *Relationships Discovery* step, navigation over the *Processed Doc Object* is performed by a syntactic dependency parser. While performing navigation, rule-based NLP is used to find tokens with *verb*-based dependencies and their associated tokens with *subject* (source concept) or *object* (target concept)-based dependencies. After finding these tokens, the existence of tokens is validated with the *Concepts Dictionary* and their corresponding lemma form is obtained. Next, for each token with verb-based dependency, a semantic similarity score is computed for word tokens which indicates the presence of a particular relationship. For example, in the sentence "employees are identified by an employee number", the "identified" word token represents an *attribute* relationship between the source concept (Employee) and the target concept (employeeNumber). Similarly, a pre-defined list of sample word tokens for "composition", "generalization", and "association" is provided manually in rule-based NLP before computing the semantic similarity score. Furthermore, we obtain "cardinalities" using the same process. For example, in the sentence "A customer can open multiple accounts at a branch", the word token "multiple" may indicate a cardinality of *zero-to-many* on the "Account" side of the *association* relationship.

In this component, the extraction of concepts, relationships, cardinalities, and their associated meta-data is driven by a set of *Qualifiers*. Each *Qualifier* is further composed of sets of *Conditions*, *Rationale*, and *Actions*. A set of conditions represents heuristics based on rule-based NLP to extract domain concepts and their meta-data. A rationale indicates a logical basis or a reason in natural language which can be presented to a modeller during traceability of modelling decisions. Finally, an action represents a set of operations required under this *Qualifer* such as creating a class or an attribute (*Descriptive Model*) and extracting the meta-data of these concepts such as sentence and word indices (*Descriptive Trace Model*). Each set of *Condition*, *Rationale*, and *Action* exists in conjunction, i.e., all its conjuncts should be true for an activated qualifier. A *Qualifier* is activated when all the *conditions* in the set are true. This will further perform all the operations in the *Actions* set and associate them with the set of *Rationale*. The *Activated Qualifiers* results in two artifacts—a *Descriptive Model* and a *Descriptive Trace Model*. The *Descriptive Model* represents the extracted concepts and their relationships along with the cardinalities on the source and target sides of a relationship, based on the actions performed under the *Activated Qualifiers*. The *Descriptive Trace Model* includes the meta-data associated with the extracted model elements and the problem description in addition to the similarity scores. We further explain the *Descriptive Trace Model* in Sect. 4.7. Though pre-trained spaCy's machine learning models are used to obtain the semantic similarity score, the direct application of rule-based NLP makes all the steps in this component dominantly as NLP-driven.

4.3 Predictive component (C)

We call this component predictive because it aims to improve the *Descriptive Model* based on the predictions of pre-trained ML models. In this component, we use machine learning techniques to obtain hints about the participating concepts and their relationships which are either difficult or not possible using rule-based NLP alone. First, in the *Concepts Clustering* step, all the concepts from the *Concepts Dictionary* are considered as primary concepts one by one and all the sentences where the primary concept exists are clustered together to form a *Concept Cluster*. Next, in the *Sentences Encoding* step, *Encoded Sentences* are generated for every *Concept Cluster* where the primary concept is paired with all other concepts from the *Concepts Dictionary* (secondary concepts) at the sentence level and special tokens are embedded around them to mark the start and the end of a concept. These special tokens are position indicators of a concept to machine learning models. For example, for the sentence "Supervisors are assigned to temporary employees", clusters for "Supervisor" and "TemporaryEmployee" concepts are created. The cluster for "Supervisor" as a primary concept and "TemporaryEmployee" as a secondary concept generates one *Encoded Sentence*—"<e1> Supervisors </e1> are assigned to temporary <e2> employees </e2>. On the other side, one possible *Encoded Sentence* in the cluster for "TemporaryEmployee" (primary concept) with "Supervisor" as a secondary concept is—"Employees can be temporary employees or permanent employees. <e2> Supervisors </e2> are assigned to temporary <e1> employees </e1>". While forming *Encoded Sentences*, we only place tags around the base word (employees) and not the whole noun chunk (temporary employees) because neighbouring words in a noun chunk are often there just to describe the noun word (concept entity).

The *Encoded Sentences* from the previous step are then processed in the *Word Embeddings Preparation* step. In this step, the semantic information about concepts and other words is obtained by using pre-trained Glove word embeddings [31] from the *Knowledge Base*. These word embeddings provide semantic-laden word representations in the form of real-valued vectors. The output from this step consists of *Word Embedding Vectors* for concepts (primary and secondary) and a *Word Embedding Matrix* $E_k^{n_k*d}$ for each encoded sentence, where n_k denotes the number of constituent words in the k^{th} sequence and d indicates the dimension of each word vector. In the *Classification for Concepts* step, concept category (*class* or *attribute*) and concept type (*class* for *class* concept and *string*, *float*, *enumeration*, *date*, *time*, or *integer* for *attribute* concept) are predicted using binary and multi-class classifiers, respectively for the given *Word Embedding Vectors*. Though the type of class concept is not used in a domain model, we keep this category to reduce the number of none type instances, i.e., when a concept is not an attribute. Also, the prediction of class type from this multi-class classifier can be combined with the prediction results of the concepts category binary classifier to model a concept as a class with higher confidence. In the *Classification for Relationships* step, pre-trained BiLSTM neural network models are used to predict relationships, e.g., association and composition, or the presence of a modelling pattern, e.g., Player-Role, and cardinalities of association relationships, e.g., zero-to-many and zero-to-one, for the given *Word Embedding Matrices*. The supervised machine learning classifiers are trained with *labeled concepts category* data and *labeled concepts type* data from the *Knowledge Base* for predicting the concepts category and concepts type, respectively. Furthermore, BiLSTM neural network models are trained with *labeled encoded sentences* data from the *Knowledge Base* for predicting the relationships or patterns and cardinalities.

Similar to the *Descriptive Component*, the *Model Preparation* step in the *Predictive Component* represents a set of *Qualifiers* which are further composed of a set of *Conditions*, *Rationale*, and *Actions*. A condition for a *Predictive Model* represents heuristics which consider all the prediction results and their probability scores to determine if a concept in the set of candidate domain concepts (which is provided by the *Descriptive Model*) is qualified to be modelled as a class or an attribute concept. Furthermore, a condition also determines if a predicted relationship and its predicted cardinalities are qualified to be included in the *Predictive Model*. For example, for the predicted relationship "composition", it is necessary that the predicted category for both primary and secondary concepts is class. Furthermore, the cardinality on the composite (whole) side of this relationship should be either *zero-to-one* or *one*. The definitions for the *Rationale*, *Action*, and *Activated Qualifier* remain the same as mentioned in Sect. 4.2 for the *Descriptive Component*.

The *Activated Qualifiers* result in two artifacts—a *Predictive Model* and a *Predictive Trace Model*. The *Predictive Model* represents the predictions for concepts category, relationships, and cardinalities on the source and target sides of a relationship, based on the actions performed under the *Activated Qualifiers*. The *Predictive Trace Model* includes the meta-data associated with the prediction results such as the probability scores, encoded sentences, and references between them. We further explain the *Predictive Trace Model* in Sect. 4.7.

In this component, the *Concepts Clustering* and *Sentences Encoding* steps represent processes which prepare data for ML. The *Word Embeddings Preparation*, *Classification for Concepts*, and *Classification for Relationships* steps use ML techniques or models for generating predictions. Finally, the *Model Preparation* step represents a process which uses *Qualifiers* to merge all the prediction results to generate the *Predictive Model* and the associated *Predictive Trace Model*.

4.4 Analysis and decision making component (D)

A *Descriptive Model* from the *Descriptive Component* abstracts a problem description where domain concepts and the relationships in which they participate are generated dominantly using rule-based NLP. On the other hand, a *Predictive Model* from the *Predictive Component* represents predictions of ML models for concepts category, relationships, and cardinalities on the source and target sides of a relationship. The results from NLP and ML techniques are now combined in this component to generate a final domain model. In this component, we use Algorithm 1 to create clusters of model extraction results based on the candidate concepts obtained from the *Descriptive Component*. The clusters obtained from Algorithm 1 are then evaluated using Algorithm 2 based on certain pre-computed decision threshold values to obtain *Evaluated Model Clusters*.

Algorithm 1: Model Clustering

```
1  Function Clustering ($cc, dm, pm$);
   Input  : Concepts Dictionary $cc$
   Input  : Descriptive Model $dm$
   Input  : Predictive Model $pm$
   Output: Model Clusters $mc$
2  $l_d \leftarrow$length($dm$), $l_p \leftarrow$length($pm$), $l_c \leftarrow$length($cc$) ;
3  $mc_{dm} \leftarrow \emptyset$, $mc_{pm} \leftarrow \emptyset$ ; /* Initialize Data Frames */
4  for $i \leftarrow 0$ to $l_c$ by 1 do
5      for $j \leftarrow 0$ to $l_d$ by 1 do
6          if $cc[i] == dm[j].source$ or $cc[i] == dm[j].target$ then
7              $mc_{dm}$.append($dm[j].relationship$);
8          end
9      end
10     for $k \leftarrow 0$ to $l_p$ by 1 do
11         if $cc[i] == pm[k].source$ or $cc[i] == pm[k].target$ then
12             $mc_{pm}$.append($pm[k].relationship$);
13         end
14     end
15     $mc$.append($cc[i]$,$mc_{dm}$,$mc_{pm}$) ;
16     $mc_{dm} \leftarrow \emptyset$, $mc_{pm} \leftarrow \emptyset$ ;
17 end
```

Algorithm 1: First, the *Concepts Dictionary*, the *Descriptive Model*, and the *Predictive Model* are used as inputs for creating their model clusters in the *Model Clustering* step. As shown in Algorithm 1, a candidate concept is considered from the *Concepts Dictionary* (L.4-17) to create *Descriptive Model Clusters* mc_{dm} and *Predictive Model Clusters* mc_{pm}. For example, the *Descriptive model dm* has relationships *(Department, composition, Employee)* and *(Employee, attribute, name)* and the *Predictive model pm* has relationships *(Department, association, Employee)*. In this case, the *Concepts Dictionary* consists of three concepts *(Department, Employee, Name)*. Next, we consider each concept from this *Concepts Dictionary* one by one to check if it is present in the *Descriptive Model dm* (L.6) or in the *Predictive Model pm* (L.11). If a concept is present in either the source or target of these models, then the relationships are added to their corresponding clusters, i.e., the *Descriptive Model Cluster* mc_{dm} (L.7) and the *Predictive Model Cluster* mc_{pm} (L.12). For the previous example, the *Employee* concept is present in both the *Descriptive Model* and the *Predictive Model*. Therefore, for the *Employee* concept, mc_{dm} is comprised of *(Department, composition, Employee)* and *(Employee,*

attribute, name) relationships and mc_{pm} is comprised of the *(Department, association, Employee)* relationship. We create a map for each concept and its corresponding data frames for its *Descriptive Model Cluster* mc_{dm} and *Predictive Model Cluster* mc_{pm} (L.15). In the end, we obtain *Model Clusters* mc which includes maps of all the concepts (L.17).

Algorithm 2: Evaluation of Model Clusters

```
1  Function Evaluation ($mc, dtr_{dm}, dtr_{pm}, dtc_{dm}, dtc_{pm}$);
   /* $dm, pm$ ≡ Descriptive and Predictive Models */
   Input  : Model Clusters $mc$
   Input  : Set of Relationship Thresholds $dtr_{dm}, dtr_{pm}$
   Input  : Set of Cardinalities Thresholds $dtc_{dm}, dtc_{pm}$
   Output: Evaluated Model Clusters $emc$
2  $l_c \leftarrow length(mc)$ ;
3  for $i \leftarrow 0$ to $lc$ by 1 do
4      $sst_r^i \leftarrow$ same source and target in $mc_{dm}^i$ and $mc_{pm}^i$ ;
5      for $j \leftarrow 0$ to $length(sst_r)$ by 1 do
6          $rel^j \leftarrow mc.rel^j$.max_diff $((mc_{dm}^i.rel_{prb}^{jx}, dtr_{dm}^{jx}), (mc_{pm}^i.rel_{prb}^{jy}, dtr_{pm}^{jy}))$ ;
7          $emc_i$.append($rel^j$); /* x,y ≡ Categories */
8      end
9      $dst_r^i = (mc_{dm}^i \cup mc_{pm}^i) \setminus sst_r^i$;
10     for $k \leftarrow 0$ to $length(dst_r^i)$ by 1 do
11         if $dst_{r_k}^i == mc_{dm}^i.rel$ and $mc_{dm}^i.rel_{prb}^{k_x} > dtr_{dm}^{k_x}$ then
12             $emc_i$.append($mc_{dm}^i.rel^k$);
13         end
14         if $dst_{r_k}^i == mc_{pm}^i.rel$ and $mc_{pm}^i.rel_{prb}^{k_y} > dtr_{pm}^{k_y}$ then
15             $emc_i$.append($mc_{pm}^i.rel^k$);
16         end
17     end
18     for $t \leftarrow 0$ to $length(emc_i)$ by 1 do
19         $r^t \leftarrow emc_i.rel^t$;    /* u,v ≡ Categories */
           /* ent ≡ source and target entities */
20         if $r^t$ in $mc_{dm}^i$ and $mc_{dm}^i.rel^t.ent.cd_{prb}^{t_u} > dtc_{dm}^{t_u}$ then
21             $emc_i.rel^t.ent.cd = mc_{dm}^i.rel^t.ent.cd$;
22         end
23         else if $r^t$ in $mc_{pm}^i$ and $mc_{pm}^i.rel^t.ent.cd_{prb}^{t_v} > dtc_{pm}^{t_v}$ then
24             $emc_i.rel^t.ent.cd = mc_{pm}^i.rel^t.ent.cd$;
25         end
26         else
27             $emc_i.rel^t.ent.cd =$ "?";
28         end
29     end
30 end
```

Algorithm 2: The model clusters obtained from the *Model Clustering* step are then used as inputs to the *Clusters Evaluation* step. In this step, we use Algorithm 2 to evaluate model clusters for each concept ($cc[i]$).

To determine the decision threshold values used in Algorithm 2, we first plot Receiver Operating Characteristic (ROC) graphs for the *Descriptive* and *Predictive Models* in each group for the validation data. We prepare this validation data by labeling the correct category in each group. These ROC graphs are two-dimensional graphs in which the True Positive rate is plotted on the Y-axis and the False Positive rate is plotted on the X-axis to depict the relative tradeoffs between True Positives and False Positives [14]. Also, we use different decision thresholds of the *Descriptive Model* and the *Predictive Model* for each category of relationships (e.g., association and composition) and each cardinality (e.g., zero-to-one and one-to-many) as inputs. These decision thresholds represent optimal cut-off points which discriminate the positive class from the negative class. We use the Youden index (Youden's J statistic) [44,49] to calculate the decision thresholds for both the *Descriptive Model* and the *Predictive Model*. The Youden index represents the vertical distance between the diagonal line and the point on the receiver operating characteristic (ROC) curve. A point which maximizes this vertical distance indicates the decision threshold for that group.

As shown in Algorithm 2, first, we find the relationships sst_r^i which have the same *(source, target)* pair but possibly different relationship types in the *Descriptive Model* dm and the *Predictive Model* pm. In sst_r^i, i represents the i^{th} concept (L.4). Second, for each relationship in sst_r^i, we compare the difference between the probability score (semantic similarity score) of relationship x in the *Descriptive Model* ($mc_{dm}^i.rel_{prb}^{jx}$) and the decision threshold for this relationship dtr_{dm}^{jx} with the difference between the probability score of relationship y in the *Predictive Model* ($mc_{pm}^i.rel_{prb}^{jy}$) and the decision threshold for this relationship dtr_{pm}^{jy} (L.6). The relationship with the maximum difference is finally added to the *Evaluated Model Cluster* emc_i (L.7). Third, we find the remaining relationships dst_r^i by taking the complement of the *Descriptive Model Cluster* mc_{dm} and the *Predictive Model Cluster* mc_{pm} with already evaluated relationships sst_r^i for the i^{th} concept (L.9). Next, we consider each relationship in dst_r^i and check if it is present in the Descriptive Model Cluster mc_{dm} and if the probability score prb of relationship x is more than the decision threshold for this relationship, i.e., dtr_{dm}^{kx}, and add it to emc_i (L.11-13). We do the same for the predictive model and add it to emc_i (L.14-16). The result is an emc_i cluster for the i^{th} concept with relationships that have probability scores greater than their respective decision thresholds. Considering the previous example, the relationship *(Department, association, Employee)* from the *Predictive Model* is considered to be included in emc_i cluster over the relationship *(Department, composition, Employee)* from the *Descriptive Model* considering that the former relationship has a greater difference between the probability score and the decision threshold (difference score). The remaining relationship *(Employee, attribute, Name)* from the *Descriptive Model* is still included in the emc_i cluster provided that its probability score is more than the decision threshold for the *attribute* relationship.

Finally, we obtain the cardinalities on the source and target sides of the relationships which are present in the emc_i cluster (L.18-29). For each relationship in the emc_i cluster, source and target entities ent are considered individually to check if the same relationship and entity exists in the *Descriptive Model* and if the probability score $ent.cd^{t_u}_{prb}$ is more than the threshold $dtc^{t_u}_{dm}$ (L.20) and to assign the cardinality (L.21). In $mc^i_{dm}.rel^t.ent.cd^{t_u}_{prb}$, $mc^i_{dm}.rel^t$ represents the t^{th} relationship in the *Descriptive Model Cluster* mc_{dm} and $ent.cd^{t_u}_{prb}$ represents the probability score of cardinality u which exists on one side (source or target) of the t^{th} relationship. On the other side, $dtc^{t_u}_{dm}$ represents the decision threshold of the *Descriptive model* for cardinality u. We do the same for the *Predictive Model Clusters*. In this assignment of cardinalities, the *Descriptive Model* has priority over the Predictive Model as the former is the result of rule-based NLP which directly computes the similarity score of the extracted word tokens which points to a particular cardinality. In addition, rule-based NLP either finds the cardinality with good confidence score (word tokens are present) or does not find it (word tokens are absent). If none of the conditions (L.20) and (L.23) are true then we assign a question mark "?" for the cardinality so that it can be determined while interacting with a modeller. Finally, the *Evaluated Model Clusters emc* are ranked according to their difference score to generate *Ranked Model Clusters*.

Similar to the *Descriptive Component* and the *Predictive Component*, the *Model Merging* step in the *Analysis and Decision Making Component* represents *Qualifiers* which are composed of a set of *Conditions*, *Rationale*, and *Actions*. A condition represents a domain modelling policy which transforms the *Ranked Model Clusters* into a *Prescriptive Model*. Domain modelling policies refer to general rules which a modeller follows while practicing domain modelling. For example, the policy "an object is composed in at most one other object" ensures that the model slices (University, *composition*, Department), (Department, *composition*, Council), and (University, *composition*, Council) lead to a configuration where "Council" is composed only once in "University". The definitions for the *Rationale*, *Action*, and *Activated Qualifier* remain the same as mentioned in Sect. 4.2 for the *Descriptive Component*.

The *Activated Qualifiers* result in two artifacts—a *Prescriptive Model* which describes the domain model and a *Prescriptive Trace Graph* which represents the meta-data of a *Prescriptive Model* such as the source of each modelling decision (*Descriptive Model* or *Predictive Model*). We further explain the generation of *Prescriptive Trace Model* elements in Sect. 4.7.

4.5 Communication component (E)

Some conflicts may arise in the *Analysis and Decision Making Component*, e.g., cardinalities which cannot be determined deterministically. These conflicts can be checked with a modeller by putting question marks "?" in place of actual cardinalities on the relationships of the extracted domain models. The *Prescriptive Model* from the *Analysis and Decision Making Component* is further processed in the *Recommendation Component* to obtain *Exposed Configurations* which are finally transformed into a *Domain Model* (class diagram) in the *Model Transformation* step. We explain the *Exposed Configurations* in Sect. 4.6. The domain model from the *Model Transformation* step is then visualized inside the *Domain Model Box* in the *User Interface*. A modeller can either accept this extracted domain model (modeller is satisfied with the extracted domain model) or can update this model (modeller recognizes some areas in the extracted domain model which can be improved) by making modifications in the *Domain Model Box*. A modeller can also select a domain model element for tracing the rationale behind the creation of the selected element.

Every change introduced by a modeller is first processed in the *Change Categorization* step where a change is either classified as *Model Update* (to trigger interactive domain modelling), *Domain Problem Update* (to regenerate the domain model), or *Model Trace Request* (to trigger the traceability of domain modelling decisions). Finally, a modeller can communicate with DoMoBOT where each message is transmitted through the *Message Transmission* step. For example, a conversation can be initiated by the bot inside the *Chat Box* in the *User Interface* when taking confirmation from a modeller before making any pro-active change to the model based on a modeller's action or can be initiated by a modeller when seeking insights into the modelling decisions taken by the bot for the extracted domain models.

4.6 Recommendation component (F)

The *Prescriptive Model* obtained from the *Analysis and Decision Making Component* is split into multiple configurations in the *Configurations Preparation* step in the *Recommendation Component*. Multiple configurations for modelling decisions are required to enable interactive domain modelling. In addition, each configuration is also split into slices. For example, for the second sentence "Employees can be temporary employees or permanent employees" of PD_{BMS}, the *Prescriptive Model* has two model configurations: one from the *Descriptive Model*, i.e., enumeration solution for *[Employee-enumeration-(PermanentEmployee, TemporaryEmployee)]*, and the other from the *Predictive Model*, i.e., Player-Role pattern for *[Employee-(PermanentEmployee, TemporaryEmployee)]*. For this scenario, both solutions are correct and there are no conflicts. Due to the higher rank of the *Descriptive Model* configuration, the enumeration solution is used in the *Exposed Configurations* for modelling this scenario. These *Exposed Configurations* are transformed into a *Domain Model*.

During bot-modeller interactions, it may be possible that a modeller has preference over the Player-Role pattern solution to support different characteristics or behaviour of the PermanentEmployee and TemporaryEmployee roles. Therefore, the Player-Role pattern solution is also included as one possible configuration in the *Configurations Preparation* step. Moreover, while preparing these configurations, slices are created for each configuration. For example, the enumeration solution can be split into four slices—(Employee, *attribute*, role), (role, *type*, Role), (Role, *enumeration_item*, TemporaryEmployee), and (Role, *enumeration_item*, PermanentEmployee). We provide *Suggestions* to a modeller in the form of alternative configurations during interaction. We obtain these alternative configurations from the *Recommendation Discovery* step. The alternative configurations depend on the degree of closeness of the current state of the model (after a modeller's change) with the candidate configurations.

Algorithm 3: We discover the alternative configurations based on a modeller's action as shown in Algorithm 3. The algorithm considers the set of exposed configurations cfs_{exp} (the current state of extracted model), the set of possible configurations cfs_{pos} (obtained from the *Configurations Preparation* step), the model element ele changed by the modeller, the modeller action type ma_{type}, e.g., create and remove (see Table 3), and the set of activated qualifiers aq_{ele} responsible for modelling ele. The exposed configurations cfs_{exp} are removed from the possible configurations cfs_{pos} to obtain the remaining configurations cfs_{rem} (L.3). For example, a modeller removes the *association* relationship between *University* and *Department* classes from the extracted model which has possible configurations—*[University-association-Department]* ($aq_{rel_1^1}$), *[University-composition-Department]* ($aq_{rel_2^1}$), *[University-class]* ($aq_{cc_1^1}$), *[University-attribute-address]* ($aq_{rel_1^2}$), and *[University-association-Address]* ($aq_{rel_2^2}$). In these configurations, $aq_{ele_m^n}$ represents an activated qualifier modelling ele (rel for relationship and cc for concept—class or attribute) with n^{th} decision instance and m^{th} configuration instance.

On the other side, the exposed configurations represent configurations with difference scores higher (difference between probability score and decision threshold) than other configurations under the same decision instance. In this case, the exposed configurations are—*[University-association-Department]*, *[University-class]*, and *[University-attribute-address]* and the set of remaining configuration cfs_{rem} includes two configurations—*[University-composition-Department]* and *[University-association-Address]*.

Algorithm 3: Discovery of Alternative Configurations

```
1  Function ConfigurationDiscovery
   ($cfs_{exp}$, $cfs_{pos}$, $ele$, $aq_{ele}$, $ma_{type}$);
   Input  : A Set of Exposed Configurations $cfs_{exp}$
   Input  : A Set of Possible Configurations $cfs_{pos}$
   Input  : Model Element $ele$ Changed by Modeller
   Input  : A Set of Activated Qualifiers $aq_{ele}$ for $ele$
   Input  : Modeller Action Type $ma_{type}$
   Output: Alternative Configurations $cfs_{rec}$
2  $cfs_{alt} \leftarrow \emptyset$ ;         /* Initialize Alternative
   Configurations */
3  $cfs_{rem} = cfs_{pos} \setminus cfs_{exp}$ ;            /* Remaining
   Configurations */
4  $l_{rem} \leftarrow$length($cfs_{rem}$);
5  for $i \leftarrow 0$ to $l_{rem}$ by 1 do
6      if $cfs^i_{rem}.qualifier$ in $aq_{ele}$ then
7          if (category(ele) in [class, attribute] and ele in
           [$cfs^i_{rem}$.source, $cfs^i_{rem}$.target]) or (category(ele)
           == relationship and $cfs^i_{rem}$.rel == ele.rel) then
8              if $cfs^i_{rem}$.flag != "DeclinedByModeller" and
               ConfigurationValidation($cfs^i_{rem}$, ele, $ma_{type}$)
               then
9                  $cfs_{alt}$.append($cfs^i_{rem}$)
10             end
11         end
12     end
13 end
14 if length($cfs_{alt}$) > 0 then
15     return $cfs_{alt}$.sort(diff_score) ;     /* Ranking by
       Difference Score */
16 end
17 return $\emptyset$ ;          /* No Configuration Found */

18 Function ConfigurationValidation ($cf_{cur}$, ele, $ma_{type}$);
   Input  : Current Configuration $cf_{cur}$
   Input  : Model Element $ele$ Changed by Modeller
   Input  : Modeller Action Type $ma_{type}$
   Output: Boolean true or false
19 if $ma_{type}$ == remove and ele not in $cf_{cur}$ then
20     return true
21 end
22 else if $ma_{type}$ in [create, update_cardinality] and ele in
   $cf_{cur}$ then
23     return true
24 end
25 else if $ma_{type}$ == group_ungroup and category(ele) !=
   category($cf_{cur}$.ele) then
26     return true
27 end
28 else if $ma_{type}$ == update_type and type(ele) !=
   type($cf_{cur}$.ele) then
29     return true
30 end
31 else if $ma_{type}$ in [rename, update_role_name] and ele.name
   not in $cf_{cur}$.ele.synonyms then
32     return true
33 end
34 else
35     return false
36 end
```

Since the modeller removes the *association* relationship, we only evaluate the configurations instances of the decision point which models the relationship between *University* and *Department* classes (L.6). This results only in one configuration—*[University-composition-Department]* from the set of remaining configuration cfs_{rem}. We check if the model element changed by the modeller is present in this refined list of configurations. Based on the category of element, we check for concept (class or attribute) if it is present on the source or target side of a relationship or for relationship which also includes cardinality and role names (L.7) in each configuration. If the element *ele* is present and the configuration does not have any "DeclinedByModeller" flag (i.e., the candidate configuration was not rejected by the modeller in the past) (L.8) then the configuration is validated (L.18-36) before it can be included in the set of alternative configurations (L.9).

During configuration validation, we check the closeness of a configuration based on the modeller action type ma_{type}. In the above example where a modeller removes the *association* relation, ma_{type} is equal to *remove*. Therefore, each configuration in this case is validated to ensure that the removed element (*association*) is not present in the configuration (L.19–21) which represents a modelling decision for the relationship between *University* and *Department* classes. If a new element such as class or attribute is created or a cardinality is updated, then the configuration is validated to ensure that this element is present in a configuration (L.22–24) to provide suggestions to a modeller based on the slices present in this configuration. For example, for a scenario, two configurations exist which represent enumeration and generalization decisions, respectively. The first configuration has four slices—(Person, *attribute*, role), (role, *attribute_type*, Role), (Role, *enumeration_item*, TemporaryEmployee), and (Role, *enumeration_item*, PermanentEmployee). In contrast, the second configuration has three slices—(Role, *generalization*, TemporaryEmployee), (Role, *generalization*, PermanentEmployee), and (Role, *association*, Person). In a model with the exposed configuration of the enumeration solution, a modeller creates a new class *PermanentEmployee*. During configuration validation, the generalization configuration returns *true* as it has the concept *PermanentEmployee* in the form of a class. Similarly, a generalization configuration is validated when a modeller brings out the *TemporaryEmployee* outside the enumeration class *Role*, i.e., a modeller action we term ungrouping of the *TemporaryEmployee* concept. In this case, we check if the current category (enumeration item) of the changed element is not equal to the category of this element (class) which exists in the generalization configuration (L.25–27).

When a modeller updates the type of an attribute or renames a class, we check if a new type is present in the configuration (L.28–30) or the synonyms list of the element in a configuration is non-empty (L.31–33). If a configuration validation fails then we return *false* to discard the configuration (L.34–36). On the other side, we include the configuration in the set of alternative configurations (L.9) and sort the configurations by their difference score in descending order (L.14–16) so that configurations with higher difference score (confidence score) are presented first to a modeller. If no configuration is found, i.e., the length of cfs_{alt} is equal to zero, then we return an empty list (L.17). These two different scenarios—(a) alternative configurations are found (the length of cfs_{alt} is nonzero) and (b) no alternative configurations are found (the length of cfs_{alt} zero)—trigger different behaviours, i.e., presenting alternative configurations to a modeller and informing a modeller that no alternative solutions are found, respectively, in NL.

As shown in Table 3, we design a taxonomy of modeller actions by considering the potential modeller actions which our proposed bot aims to support during bot-modeller interactions. Our bot currently supports two categories of modeller actions—Model Update and Problem Update. In the future, we plan to extend this taxonomy by including more action categories which represent potential scenarios in interactive domain modelling and can be supported by our bot. Each action category is further categorized into types such as classes and into actions such as renaming a class or removing a relationship in the *Action Categorization* step. A strategy is created based on the category of modeller's actions in the *Strategies Creation* step. For example, when a modeller creates a new class then a strategy is created which checks for an alternative configuration which is close to the current state of the model (with the new class). If a potential configuration is discovered then a subsequent strategy is created to present the suggestion to the modeller and take confirmation before updating the model.

In addition, for each modeller action, we present the corresponding bot-modeller interaction scenarios in Table 3. For example, a scenario is modelled using the Player-Role pattern solution where a generalization relationship is present between superclass *Role* and its subclasses (*TemporaryEmployee* and *PermanentEmployee*) in an extracted domain model. A modeller who is interested in using the enumeration solution over the Player-Role pattern solution can initiate bot-modeller interaction by performing an action in several ways. Each time a modeller performs an action, the bot detects if the new state of the model is closer to one of the alternative configurations. One possible action can be *Removing a class* where the modeller removes the *Role* class. However, in the alternative configuration (enumeration solution), the *Role* concept is present which results in no response from the bot.

Another possible action can be *Group and Ungroup* where the modeller brings the concept *PermanentEmployee* inside the superclass *Role* (drag and drop). In this case,

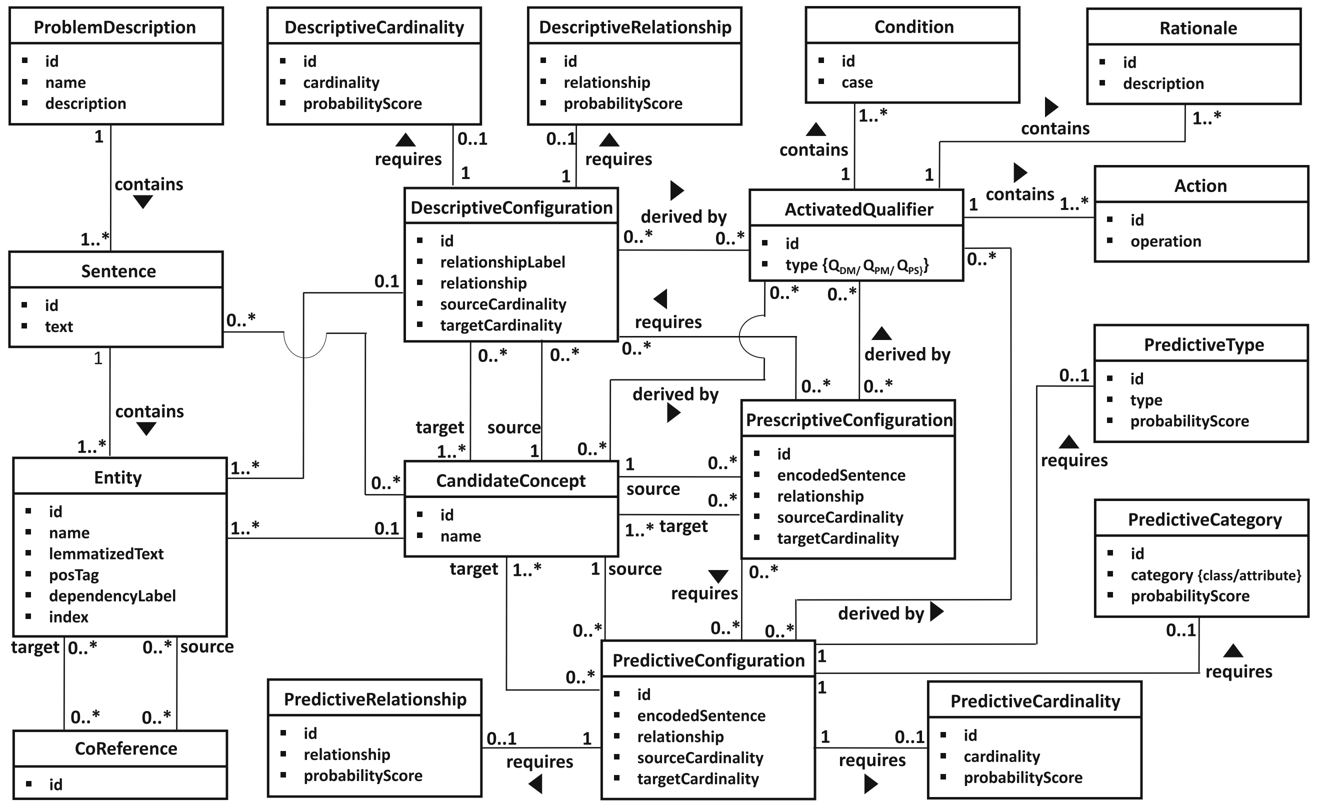


Fig. 3 Traceability Information Model (taken from [38])

the property of concept *PermanentEmployee* changes from no group (class) to group (attribute or enumeration item). With this change, the bot detects that the new state is close to another configuration (enumeration solution) and checks with modellers in the form of suggestions before updating the model (switching the configuration from Player-Role solution to enumeration solution for this scenario). When a modeller confirms or rejects a suggestion (change in the model) then the configurations or slices associated with that suggestion are marked with "AcceptedByModeller" or "DeclinedByModeller" flags, respectively. These flags can avoid recommending a modeller a rejected suggestion again.

4.7 Tracing component (G)

The knowledge about the domain model (to be built) evolves over multiple components (from Descriptive Component to Prescriptive Component). Therefore, traceability relations are required to map the knowledge state at one component to the knowledge state at another component. In our architecture, the knowledge states are represented by the three trace models—*Descriptive*, *Predictive*, and *Prescriptive Trace Models*. These models are transformed into the graphs in the *Graphs Generation* step in the *Tracing Component*. Furthermore, the *Tracing Component* integrates these three graphs and generates the *Traceability Knowledge Graph* in the *Graphs Integration* step. The *Traceability Knowledge Graph* enables modellers to trace domain modelling decisions from domain model elements to the problem description (backward traceability) and from problem description to the domain model elements (forward traceability). The *Traceability Knowledge Graph* conforms to the *Traceability Information Model (TIM)* as shown in Fig. 3. The *TIM* is comprised of two types of elements—traceable artifacts and traceability relations between these artifacts. Each type of artifact is represented by a UML class and a valid trace between two types of artifacts is represented by a relation. During the instantiation of the domain model extraction process for a problem description, we map automatically each class in the TIM to a node, the attributes of a TIM class to the properties of a node, and the relations between two classes to the node relationships in a trace graph. Each node in a graph has an *id* property which takes a unique integer. Though navigation is possible from either sides of a relationship in this graph due to its bidirectional nature, we show small arrows in the *TIM* to indicate the direction when interpreting the annotated relations.

In the case of the *Descriptive Trace Graph* for PD_{BMS}, first, we create the *ProblemDescription* node with properties—*name* (Bank Management System) and *descrip-*

Table 3 Taxonomy of modeller actions with interaction scenarios

Category	Type	Action	Interaction Scenario (i: Initial State, M: Modeller, B: Bot, f: Final State)	Rationale
Model Update	Class	Rename	i: extracted model has class *Exam* M: starts renaming this class B: provide synonym suggestions, e.g., Examination M: accepts the suggestion f: class name is changed	The new name is closer to the old name syntactically or semantically
		Remove	i: *association* between *Branch* and *Location* classes M: removes class *Location* B: checks for alternative configuration; finds one configuration B: suggests attributes - *areaCode* and *location* for class *Branch* M: accepts the suggestions f: *Branch* has attributes (*location:String* and *areaCode:Integer*)	The bot finds a configuration which is closest to the current state (after modeller's action) of the existing configuration. In new configuration, concept *location* is modelled as an attribute
		Create	i: *temporaryEmployee* is an item of the enumeration class *Role* M: creates a new class *TemporaryEmployee* B: checks for alternative configuration with *TemporaryEmployee* as a class, finds the generalization solution B: suggests the generalization solution to model this scenario M: accepts the suggestion f: modelling decision for this scenario changes to *generalization*	The bot finds the generalization solution as the closest alternative configuration to the current state (after modeller's action) of the existing configuration. In the generalization solution, concept *TemporaryEmployee* is modelled as a class
		Group and Ungroup	i: subclass *TemporaryEmployee* with superclass *Role* M: brings subclass *TemporaryEmployee* inside superclass *Role*, i.e., *TemporaryEmployee* is a part of another concept (group) B: checks for alternative configuration with concept *TemporaryEmployee* inside a class, finds enumeration B: suggests the enumeration solution to model this scenario M: accepts the suggestion f: modelling decision for this scenario changes to *enumeration*	The bot finds the enumeration solution as the closest alternative configuration to the current state (after modeller's action) of the existing configuration. In the enumeration solution, concept *TemporaryEmployee*is not modelled as a class
	Attribute	Update Type	i: attribute *weight* in class *Luggage* has type *Integer* M: updates the type of this attribute to *Float* B: checks if this attribute exists in other classes; finds an attribute *allowedWeight* in class *Seat* B: suggests the change of its type to modeller M: modeller accepts the change f: type of attributes in both classes is *Float*	Attributes *weight* and *allowedWeight* are semantically similar and often modelled with the same type
		Rename	i: extracted model has an attribute *color* M: starts renaming this attribute B: provide synonym suggestions, e.g., *shade* M: accepts the suggestion f: attribute name is changed	The new name is closer to the old name syntactically or semantically

Table 3 continued

Category	Type	Action	Interaction Scenario (i: Initial State, M: Modeller, B: Bot, f: Final State)	Rationale
		Remove	i: extracted model has attribute *result* M: removes this attribute B: checks for alternative configuration without this attribute, finds a solution with the concept *Result* modelled as a class B: suggests alternative solution to model this scenario M: accepts the suggestion f: modelling decision for this changes from attribute to class	The removed attribute is modelled as a class in another configuration
		Create	i: model has a class *Examination* with no attribute M: starts adding an attribute to this class B: checks for configuration with an attribute in this class, finds three attributes - *level*, *room*, and *type* B: suggests these three attributes to modeller M: selects attribute *level* f: class *Examination* has attribute *level*	The class *Examination* has attributes in alternative configurations such as level, room, and type
	Attribute	Group and Ungroup	i: attribute *address* in class *Customer* M: brings concept *address* outside class *Customer* (ungroup) B: checks for alternative configuration where concept *address* is not an attribute; finds the association solution B: suggests an association relationship between *Customer* and *Address* classes M: accepts the suggestion f: modelling decision for this changes to association	The bot finds the association solution as the closest alternative configuration to the current state (after modeller's action) of the existing configuration. In the association solution, concept *Customer* is modelled with multiple *Addresses*
	Relationship	Update Cardinality	i: one side of association is missing its cardinality ("?") M: provides an 0..* cardinality for a question mark B: checks if any configuration exists with the provided cardinality; finds a configuration with two relationships including the association relationship; B: suggests the other relationship to the modeller M: accepts only association relationship suggestion f: association relationship with 0..* cardinality in the model	The bot finds other configurations where the relationship has 0..* cardinality and this configuration has other relationships where cardinalities are present but absent in the extracted model.
		Update Role Name	i: association relationship has role name *registers* on one side M: starts updating the role name B: checks if synonyms exist for this role name; finds *has_registrations* M: accepts the suggestion f: role name is changed to *has_registrations*	The new role name is closer to the old name syntactically or semantically.

Table 3 continued

Category	Type	Action	Interaction Scenario (i: Initial State, M: Modeller, B: Bot, f: Final State)	Rationale
		Remove Relationship	i: relationship *Generalization* between superclass (*Role*) and subclasses (*TemporaryEmployee, PermanentEmployee*) M: removes this relationship B: checks for alternative configuration without *generalization* relationship for this scenario; finds *enumeration* B: suggests the *enumeration* solution to model this scenario M: accepts the suggestion f: modelling decision for this changes to *enumeration*	The bot finds the *enumeration* solution as the closest alternative configuration to the current state (after modeller's action) of the existing configuration. In the *enumeration* solution, there is no *generalization* relationship between the participating concepts
		Create Relationship	i: model with classes *Customer* (attribute *address*), *Address*, and *Bank*; *association* between *Address* and *Bank* classes M: creates an *association* between *Customer* and *Address* B: suggests to remove attribute *address* M: accepts the suggestion of removing attribute *address* B: removes attribute *address* of class *Customer* f: modelling decision changes from *attribute* to *association*	The bot finds the association relationship as the closest alternative configuration where concept *Customer* is modelled with multiple *Addresses*
	Modeller's Response	Reject or Accept Suggestion	i: modeller performs any of the above changes B: discovers an alternative configuration and suggests the same to modeller; M: rejects the suggestion B: mark the suggestion with "DeclinedByModeller" flag.	A modeller's rejected change is flagged "DeclinedByModeller" so that the same suggestion is not presented again. In contrast, accepted suggestion is flagged "AcceptedByModeller" to indicate the model element or decision is confirmed by a modeller.
Problem Update	Requirements	Adding Information	i: a model is extracted for a given problem description M: provides additional information B: triggers every step in Components A to E f: a new domain model is generated	With the augmented information, extracted concepts and relationships in which they participate may change. Hence, all the steps are re-triggered
		Removing Information	i: a model is extracted for a given problem description M: removes information B: triggers every step in Components A to E f: a new domain model is generated	With the removed information, extracted concepts and relationships in which they participate may change. Hence, all the steps are re-triggered

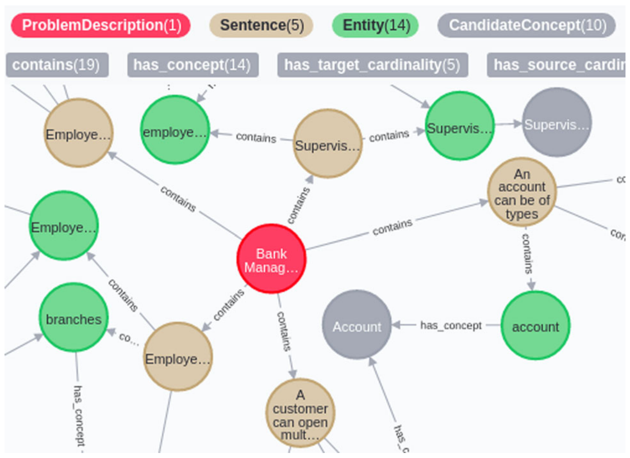


Fig. 4 An excerpt from Traceability Knowledge Graph

tion (the text of PD_{BMS}). Second, for each sentence in PD_{BMS}, we create a node *Sentence* with different properties (*id*, *text*). Third, we represent each word token in a sentence by individual *Entity* nodes with properties which includes name, index, and linguistic features such as POS tags and dependency labels for each word token. Fourth, we create nodes for references between source and target entities which are obtained from the CoReference Resolution technique. Finally, we create a node for each *ActivatedQualifer* with properties, e.g., *type* (“Q_{dm}” which represents *Activated Descriptive Model Qualifier*). These qualifiers drive the construction of concepts (*CandidateConcept*) and their corresponding relationships (*DescriptiveConfiguration*). Figure 4 shows an excerpt from the Traceability Knowledge Graph constructed for the PD_{BMS}. In this figure, we show a node for *Problem Description* (red color) which *contains* five *Sentence* nodes (brown color). Each sentence further contains nodes for *Entity* nodes (green color). In addition, we show “Account” and “Supervisor” *CandidateConcept* nodes (grey color) which are linked with the “account” and “supervisor” *Entity* nodes, respectively.

We also create corresponding *Rationale*, *Condition*, and *Action* nodes. For example, to extract domain concepts (employee and employeeNumber) from the sentence of PD_{BMS} “Employees are identified by an employee number”, the *ActivatedQualifer* node contains a *Condition* node which has a property *case* (checks if the entity exists in a noun chunk and if its *posTag* is noun-based such as ’NN’ that represents noun, singular, or mass). The corresponding *Rationale* node has a property *description* (entities in a noun chunk may represent a domain concept—class or attribute). In addition, the *Action* node has a property *operation* (create *CandidateConcept* nodes for “Employee” and “employeeNumber” and create their relationships with corresponding entities). A similar process is followed for the *DescriptiveConfiguration* which further requires the relationships and cardinalities from the *DescriptiveRelationship* and the *DescriptiveCardinality* nodes, respectively. The latter nodes also have a *probabilityScore* property which represents a real-value between 0 and 1.

A similar process is followed to obtain the *Predictive Trace Graph* which is comprised of nodes—*Predictive Configuration*, *PredictiveRelationship*, *PredictiveCardinality*, *ActivatedQualifier* with property *type* (“Q_{pm}” which represents *Activated Predictive Model Qualifier*) and corresponding *Condition*, *Rationale*, and *Action* nodes. In contrast, the *ActivatedQualifier* nodes with properties *id* and *type* (‘Q_{ps}” which represents *Activated Prescriptive Qualifier*) derive the *PrescriptiveConfiguration* nodes of the *Prescriptive Trace Graph*. The other nodes in this graph are *Condition*, *Rational*, and *Action*. We do not include *Decision Thresholds* and *Ranked Model Clusters* in the *TIM* as they represent the intermediate results while creating prescriptive configurations. The *PrescriptiveConfiguration* node requires *DescriptiveConfiguration* and *PredictiveConfiguration* nodes for models evaluation.

All nodes (artifacts) in the *Prescriptive Trace Graph* are traceable to the nodes (artifacts) of the *Descriptive Trace Graph* and the *Predictive Trace Graph* through *CandidateConcept* nodes (artifacts). We call the traceability relations across these three graphs *cross-graph traceability*. The *Modelling Decisions Traceability* exist between domain model elements and the *Traceability Knowledge Graph* through the *Activated Qualifiers* as their actions create domain model elements as well as the *Traceability Knowledge Graph* elements. A modeller can initiate a *Model Trace Request* by selecting a domain model element for tracing the rationale behind the creation of the selected element. This enables the modeller to gain insights into the modelling decisions of the selected domain model element. Each request is transformed into a *Trace Graph Query* in the *Strategies Creation* step in the *Recommendation Component*. Finally, each *Trace Graph Query* is transformed into a native query in the *Native Query Generation* step to retrieve the artifacts and relations associated with the selected model element. The *Trace Results* are then transformed into a *Model Trace Response* so that they can be presented to a modeller in the form of highlighted sentences or words in the problem description. Also, the *Rationale* associated with the selected model element is used as a template in the form of additional informative messages in NL to provide further insights into the modelling decisions to a modeller.

4.8 Query answering component (H)

A modeller can communicate with the bot by writing messages in NL. Each message is then transmitted as a *Query* to the *Query Answering Component*. In the *Intent Classi-*

fication step, we use the RASA[5] NLU (Natural Language Unit) pipeline which includes pre-trained models. These models further use the spaCy library to load pre-trained language models so that each word can be transformed into word embeddings. One of the models in this pipeline is Dual Intent Entity Transformer (DIET) which is used for intent classification and entity extraction for each NL message. DIET classifies the messages into one or multiple pre-defined intents. We manually classify some possible intents into discrete categories which our bot aims to support while facilitating bot-modeller interactions and traceability of modelling decisions, as shown in Table 4. Bot-modeller interaction for updating the model includes intent classes - *Offer Suggestions*, *Accept Suggestion*, *Decline Suggestion*, *Query Configuration*, and *Provide Information*. On the other side, traceability of modelling decisions includes intent classes - *Find Rationale*, *Explain Rationale*, *Show Examples*, *Trace Backward*, and *Trace Forward*. In the future, we plan to extend the classification strategy (see Table 4) to cover a wider range of use case scenarios which could be possible during interactive and traceable domain modelling.

We prepare training data for these intent classes by creating possible examples and labeling their intent class manually. For example, *Accept Suggestion* is predicted during intent classification if a modeller's message is similar to examples—"yes", "ok", and "make the change". In addition, a modeller's selection of the "Yes" option (when the bot asks a question in "Yes" or "No") can also trigger this intent. Moreover, an intent *Trace Backward* can be initiated by a modeller using NL messages in addition to selecting a model element.

An intent can also be forced (*Triggered Intent*) from the *Recommendation* and *Tracing* components. For example, when a modeller selects a modelling element for tracing its modelling decision, the *Tracing* component can forcefully trigger the *Explain Rationale* intent so that the *description* property of the *Rationale* node associated with the selected modelling element can be presented to the modeller in NL. In addition, the *Recommendation Component* can forcefully trigger the *Offer Suggestions* intent so that a confirmation can be taken from a modeller before updating the model proactively based on an alternative configuration (*Suggestions*) found during bot-modeller interaction. During intent classification, multiple intents can also be discovered. For example, *Explain Rationale* and *Show Examples* intents can be predicted simultaneously so that the bot provides insights into the modelling decision of an entity, e.g., class and relationship using NL messages as well as using examples from the training data. We use examples from the training data because the ML models in the *Predictive Component* also use this data to generate predictions for a new problem description. The training data may represent diverse scenarios for modelling decisions.

The outputs of the *Intent Classification* step are *Predicted Intent(s)* and *Extracted Entities* (if any). The *Predicted Intents* and *Extracted Entities* are then processed in the *Actions Generation* step to generate corresponding *Actions*. For example, for a predicted intent *Trace Backward*, multiple actions are generated—(1) trigger a *Trace Query* for the *Tracing Component*, (2) Retrieve *Sentence* and *Entity* nodes from the *Traceability Knowledge Graph* (*Trace Graph Results*), and (3) Highlight the sentences or word tokens based on the *Trace Graph Results*. An action can also be triggered for the *Recommendation Component*. For example, for a predicted intent *Query Configuration*, multiple actions are generated—(1) Send query to *Recommendation Component* to check for alternative configurations for a modelling decision (*Request*), (2) If a configuration is found then trigger another intent *Offer Suggestions* to provide alternative configurations to the modeller (*Suggestions*). Otherwise, trigger *Inform User* intent to inform that no suggestion is found. We do not include intents such as *Inform User* and *Greet User* in our classification strategy for modeller intents (see Table 4) because they are general intents (not specific to domain modelling) which involve sending direct NL messages to the user. The *Intent Classification* step is based on the direct application of ML techniques while the *Action Generation* step represents a general process.

5 Experiment design

In this section, we first present our tool and then discuss the three research questions that our evaluation addresses.

5.1 DoMoBOT tool

To help modellers in quickly prototyping domain models for given problem descriptions and to facilitate bot-modellers interactions in addition to providing traceability of modelling decisions, we build the DoMoBOT tool, a domain modelling bot. We instantiate the components of the architecture proposed in Sect. 4 and implement them in the form of a web-based tool. In this tool, the *Pre-processing (A)*, *Descriptive (B)*, *Predictive (C)*, *Analysis and Decision Making (D)*, *Recommendation (F)*, *Tracing (G)*, and *Query Answering (H)* components in addition to the *Knowledge Base* represent the backend of our tool. On the other side, the *Communication Component (E)* and the *User Interface* represent the frontend of our tool. We use Python[6] and React JS[7] to build the back-

[5] https://rasa.com/.

[6] Python 3.6.12—https://www.python.org/.

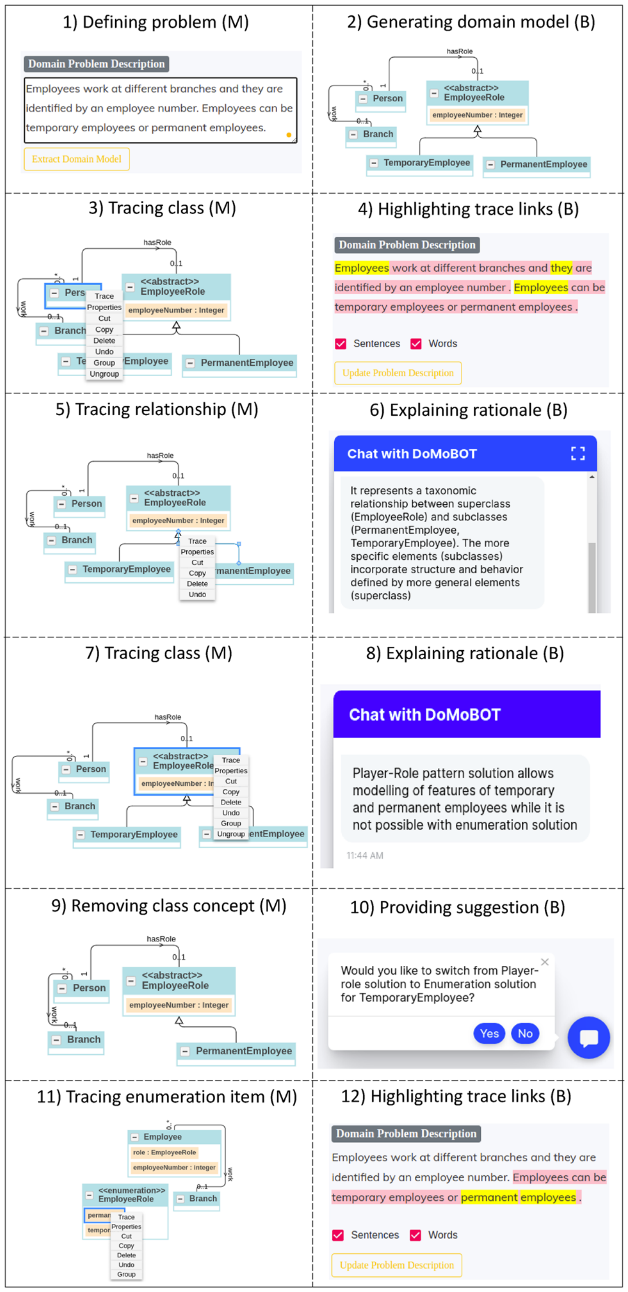
[7] React 17.0.1—https://reactjs.org/.

Table 4 Classification Strategy for Modeller Intents

Intent Class	Examples (B: Bot; M: Modeller)	Generated Action	Rationale
Offer Suggestions	B: Do you want to switch from generalization solution to enumeration solution for TemporaryEmployee?; B: Do you want to add the attributes (address, name, areaCode) to Bank class?	Send confirmation request/suggestions to modeller.	Based on modeller's action, bot finds alternative configurations. Before updating the model, bot takes confirmation. This intent is triggered by the *Recommendation Component*.
Accept Suggestion	M: Yes; M: I agree with this suggestion; M: Make this change; M: Add address and name to the Bank class; M: <<selecting>> Yes option	Update domain model; Assign the "AcceptedByModeller" flag to the accepted configuration.	After modeller's confirmation, the domain model needs to be updated based on the accepted configuration (suggestions).
Decline Suggestion	M: No; M: I do not want this change; M: Do not add these attributes; M: <<selecting>> No option	Do not update the domain model; Assign the "DeclinedByModeller" flag to the rejected configuration.	A configuration (suggestion) rejected by modeller is assigned a "DeclinedByModeller" flag to avoid presenting a rejected suggestion again to modeller.
Query Configuration	M: Any other way to model concept PartTimeStudent?; M: Any other solution for the PartTimeStudent role?	Query *Recommendation Component* to find alternative configurations for the entity present in the message.	A modeller can obtain alternative configurations which models *PartTimeStudent* roleby sending messages in NL.
Provide Information	B: Can you provide more information for concept FullTimeEmployee? B: Can you provide missing cardinalities?	Send message to modeller.	During model extraction, a conflict may arise in modelling a concept or some concepts are not modelled, e.g., cardinalities due to low confidence score.
Find Rationale	M: Why is the Employee class created?; M: I do not understand the reason behind the class employee; M: Explain to me the rationale for Employee	Query *Tracing Component* for the entity present in the message. Retrieve the Rationale Nodes from the *Traceability Knowledge Graph*.	A modeller can trace modelling decisions for a domain model element using messages in NL.
Explain Rationale	B: It represents a taxonomic relationship between a superclass VehicleType and subclasses (Automatic, Manual). The more specific elements (subclasses) incorporate structure and behavior defined by more general elements (superclass).	Send the description of each retrieved rational node to the modeller	When modeller traces the modelling decision of a concept, the rational nodes associated with that concept are retrieved. The description property of these nodes is presented to modeller in NL.
Show Examples	M: Show me examples for generalization; M: Explain generalization with examples; B: Do you need more information for generalization relationship?	Query *Tracing Component* for the entity present in the message. Retrieve examples from the Knowledge Base for the entity.	A modeller may ask for examples that explain a concept or relationship better. For example, for the detected entity generalization, examples from training data can be obtained.
Trace Backward	M: Show me sentences responsible for class University; M: Which words contribute to modelling University class	Query *Tracing Component* for the entity (concept) present in the message. Highlight the results from the Traceability Knowledge Graph (Sentence and Entity nodes).	A modeller may write messages in NL to retrieve traceability links between domain model elements and problem description. The results are presented in the form of highlighted sentences and/or words.
Trace Forward	M: Which concepts are modelled from the information provided in the first sentence?; M: Show me concepts from the first three sentences	Query *Tracing Component* for the entity (sentence) present in the message. Explain the results from the Traceability Knowledge Graph (e.g., CandidateConcept) in NL.	A modeller may write messages in NL to retrieve traceability links between domain model elements and problem description. The results (e.g., concepts name) are presented in the chatbox.

end and the frontend, respectively. In addition, we use Neo4j[8] which is an open-source native graph knowledge database to persist the artifacts of the *TIM* and their relations in the form of the *Traceability Knowledge Graph*. To build the *Query Answering (H)* component and prepare ML models in the *Knowledge Base*, we use RASA[9] which is an open source ML framework for automated text and voice-based conversations. Furthermore, we use the spaCy library[10] and Glove word embeddings [31] across various components and for the *Knowledge Base* in our proposed architecture. Finally, we integrate our backend and frontend using the Django framework[11] and visualize domain models (class diagrams) in the *User Interface* using the GOJS framework[12].

Figure 5 illustrates with screenshots some possible interactions between the bot (B) and modeller (M) inside our proposed tool, DoMoBOT.


1) Defining problem (M)
Domain Problem Description
Extract Domain Model
2) Generating domain model (B)
hasRole
0..1
1
0..*
work
0..1
Person
<<abstract>> EmployeeRole
employeeNumber : Integer
Branch
TemporaryEmployee
PermanentEmployee
3) Tracing class (M)
Trace
Properties
Cut
Copy
Delete
Undo
Group
Ungroup
4) Highlighting trace links (B)
Domain Problem Description
Sentences
Words
Update Problem Description
5) Tracing relationship (M)
6) Explaining rationale (B)
Chat with DoMoBOT
7) Tracing class (M)
8) Explaining rationale (B)
Chat with DoMoBOT
11:44 AM
9) Removing class concept (M)
10) Providing suggestion (B)
Yes
No
11) Tracing enumeration item (M)
Employee
role : EmployeeRole
employeeNumber : integer
<<enumeration>> EmployeeRole
Branch
12) Highlighting trace links (B)
Domain Problem Description
Sentences
Words
Update Problem Description

Employees work at different branches and they are identified by an employee number. Employees can be temporary employees or permanent employees.

It represents a taxonomic relationship between superclass (EmployeeRole) and subclasses (PermanentEmployee, TemporaryEmployee). The more specific elements (subclasses) incorporate structure and behavior defined by more general elements (superclass)

Player-Role pattern solution allows modelling of features of temporary and permanent employees while it is not possible with enumeration solution

Would you like to switch from Player-role solution to Enumeration solution for TemporaryEmployee?

Fig. 5 An illustration of DoMoBOT for Bot (B) - Modeller (M) Interactions

- **Step (1):** modeller provides a problem description in NL and clicks “Generate Domain Model” button.
- **Step (2):** bot extracts domain model automatically from the given problem description.
- **Step (3):** modeller selects “Person” class and uses the context menu to select the *Trace* option for retrieving the trace links for the selected model element.
- **Step (4):** bot queries the *Traceability Knowledge Graph* and presents the results in the form of highlighted sentences and words.
- **Step (5):** modeller selects the “generalization” relationship and uses the context menu again to select the *Trace* option for getting insights into this modelling decision.
- **Step (6):** bot explains the modelling decision rationale for the selected element in NL.
- **Step (7):** modeller selects the “EmployeeRole” class which is participating in the Player-Role pattern. After selecting this element, modeller selects the *Trace* option from the context menu.
- **Step (8):** bot explains the modelling decision rationale for the selected model element in NL.
- **Step (9):** modeller is interested in modelling the scenario using the enumeration solution. Therefore, modeller performs class-remove action, i.e., removal of the “TemporaryEmployee” class.
- **Step (10):** bot detects the change (action performed by modeller) and starts searching for an alternative configuration which is closest to the current state of the model (TemporaryEmployee concept as a class does not exist now). Bot finds enumeration solution as an alternative configuration and offers this suggestion to modeller. When modeller accepts the suggestion by selecting the “Yes” option, the bot updates the model automatically as shown in Step (11).
- **Step(11):** modeller selects “permanent” concept (enumeration item) and uses the context menu to select the *Trace* option for retrieving the trace links for the selected model element.

[8] Neo4j 4.1.10—https://neo4j.com/.

[9] RASA 2.1.3—https://rasa.com/.

[10] spaCy 2.3.5—https://spacy.io/.

[11] Django 2.2.7—https://www.djangoproject.com/.

[12] GOJS 2.1.35—https://gojs.net/latest/index.html.

– **Step (12):** bot queries the *Traceability Knowledge Graph* and presents the results in the form of highlighted sentences and words.

5.2 Dataset construction process

We use a set of domain problem descriptions which represent systems such as Online Rideshare System, Flight Reservation System, Sports League Management System, and Election Management System. These problem descriptions represent modelling exercises which are created by instructors and teaching assistants in software modelling courses at universities. We split this set of problem descriptions in two sets - training dataset of problem descriptions (PD_{train}) and test dataset of problem descriptions (PD_{test}). PD_{train} is used for training ML models and enhancing the rule-based NLP system for domain model extraction in our previous work [36,37,40]. In addition to PD_{train}, we collect a corpus of noun words from WordNet [26,27]. Next, we manually label their category (class or attribute) as well as their type (string, integer, float, date, time, enumeration in the case of attribute concept and class in the case of class concept) [37]. Finally, we remove the none cases to obtain the commonly used noun words along with their labels. This labeled data is used for training binary and multi-class classifiers in the *Predictive Component*.

Furthermore, we use a semi-automated process for creating training data for BiLSTM models for the prediction of relationships and cardinalities [36,40]. In this semi-automated process, we first extract sequences automatically and embed special tokens around primary and secondary concepts. Next, we manually identify substitutable fragments and their corresponding equivalent fragments for these sequences. For example, for a sequence *"students are assigned supervisors"*, substitutable fragments are *students* and *supervisors*. In this case, one possible combination of equivalent fragments is (*employees*, *managers*). In addition, we can introduce some variations in the original sequences (sentences) to include more scenarios, e.g., "students work under the guidance of supervisors". For both PD_{train} and PD_{test}, we label the data based on the ground truth domain model solutions which we manually create for respective problem descriptions. These ground truth domain models represent ideal solutions of problem descriptions. We also consider multiple equivalent correct solutions for each problem description while creating ground truth domain models.

We further randomly split PD_{test} in two sets—a validation dataset PD_{valid} and an evaluation dataset PD_{eval}. PD_{valid} is used for calculating decision threshold values for the *Descriptive* and *Predictive Components* in each group, e.g., concept category and relationship predictions. In this paper, PD_{eval} is comprised of 12 problem descriptions. These problem descriptions were never seen or touched while performing the design or development activities for our tool. We use PD_{eval} for evaluating the effectiveness of the bot's responses during bot-modeller interactions.

5.3 Research questions

In this section, we present the three research questions that our evaluation addresses for the effectiveness and usefulness of our tool in facilitating bot-modeller interactions. We present our motivation and methodology for each research question in this section.

RQ1 (Effectiveness): How effective are the bot's responses in providing suggestions and updating domain models?

Motivation: In our previous work, we evaluate the accuracy of extracted domain models [36,37,40] and the effectiveness of traceability links [38]. In this paper, we focus on bot-modeller interactions where our bot discovers alternative configurations and interacts with a modeller in different ways—presenting suggestions to a modeller in NL, taking modeller's confirmation for the offered suggestions, and updating the extracted domain model after the acceptance of offered suggestions. These interactions are based on the results of *Recommendation* and *Query Answering Components* in addition to the communication between these two components. For example, when an alternative configuration is found in the *Recommendation Component*, an intent *Offer Suggestions* is triggered by the *Recommendation Component* so that the suggestion can be communicated to a modeller through the *Query Answering Component*. For each intent, one or multiple corresponding actions are performed by the *Query Answering Component*. Therefore, we assess the performance of these components by evaluating the effectiveness of the bot's responses during bot-modeller interaction.

Methodology: We evaluate the effectiveness of the bot's responses in three categories—*Found Configurations*, *Offered Suggestions*, and *Updated Domain Models*. The *Found Configurations* category represents configurations which are discovered when the model state changes (modeller actions are performed). The *Offered Suggestions* category indicates the suggestions offered by the bot to a modeller in NL when an alternative configuration is discovered. Finally, the *Updated Domain Model* category represents the changes performed by the bot in the extracted domain model when the modeller accepts the offered suggestions. Furthermore, we perform actions from Table 3 which are currently fully supported by our proposed tool—Remove Class, Create Class, Group and Ungroup Class, Remove Attribute, Create Attribute, Group and Ungroup Attribute, Update Cardinality, Remove Relationship, Reject or Accept Suggestion, Adding Information and Removing Information. While performing these actions on decision points in the extracted domain models for each problem description, the bot discovers and presents the closest configurations for the modified decision point. To answer our research question, we check whether the offered configurations are relevant or not based on the ground truth domain models. Moreover, we include various variations of the ground truth domain model created by modellers or instructors other than the authors.

To evaluate the effectiveness of the bot's responses in these three categories, we use PD_{eval} and their corresponding ground truth domain models. First, we manually identify the decision points in the ground truth domain model for each problem description in PD_{eval}. These decision points represent the concepts such as class and relationship which can be modelled with two or many configurations (different modelling decisions). For example, the decision point for modelling multiple roles (temporary and permanent) of "Employee" includes two configurations—a configuration using the enumeration solution and a configuration using the Player-Role pattern solution. In the enumeration solution, the "temporary" and "permanent" roles are modelled as enumeration items. In contrast, these roles are modelled as subclasses in the Player-Role pattern solution. Second, we use our tool and extract the domain model from each problem description and check if the decision points (identified from the ground truth domain models) are also present in the extracted domain models.

Third, for every decision point present in an extracted domain model, we calculate the total number of possible configurations manually which are relevant to the decision points, i.e., True Positives (TP) + False Negatives (FN). We consider only those decision points from ground truth domain models which are also present in the extracted domain models to avoid introducing inaccuracies in our evaluation for bot-modeller interaction to some extent. These inaccuracies are the result of the model extraction process, i.e., some decision points (model elements) of the ground truth domain models are missing in the extracted domain models. Fourth, we perform various modeller actions (see Table 3) on every decision point present in an extracted domain model. The modeller actions are performed on the model elements which represent a decision point. Fifth, we calculate the total number of configurations which are discovered when the state of extracted domain model changes (modeller actions performed on decision points), i.e., TP + False Positives (FP). Sixth, we calculate the total number of discovered configurations which are relevant to the decision point, i.e., TP. Finally, we calculate the precision[13] (Pr), recall[14] (Re), and F1[15] scores for the *Found Configurations* category:

$$Pr = \frac{TP}{TP + FP} \quad Re = \frac{TP}{TP + FN} \quad F1 = 2\,\frac{Pr * Re}{Pr + Re}$$

Similarly, we obtain precision, recall, and F1 scores for *Offered Suggestions* and *Updated Domain Models* categories. In the case of the *Offered Suggestions* category, the total number of possible configurations for a decision point is equal to the total number of offered suggestion (TP + FN). Furthermore, we calculate the total number of suggestions which are generated by the bot in the form of informative messages in NL inside the chat widget (TP + FP) and the total number of relevant suggestions generated by the bot (TP). In the case of the *Updated Domain Models* category, we calculate the total number of model elements such as classes and relationships for all possible configurations of each decision point (TP + FN). Next, we calculate the total number of elements for all configurations of every decision point one by one, which are modelled by the bot after the acceptance of the bot's suggestions (TP +FP). To measure this efficiently, we accept all the suggestions offered by the bot from the *Offered Suggestions* category. Finally, we calculate the total number of correctly modelled elements by the bot for all configurations of every decision point (TP).

RQ2 (Performance): Does our tool discover alternative configurations and generate suggestions in practical time?

Motivation: Problem descriptions may represent a large number of domain concepts which can be modelled with multiple configurations. In addition, each configuration may include multiple slices, e.g., the Player-Role pattern includes slices such as the *generalization* relationship between *Role* (abstract superclass) and multiple roles (subclasses) and the *association* relationship between *Role* and *Player* classes. Moreover, our bot aims to assist modellers in obtaining the desired configuration soon after the modeller's action has changed the extracted model, bringing it closer to one of the alternative configurations. Also, with the discovery of alternative configurations, the corresponding suggestions should be presented to modellers quickly. Therefore, it is important to study whether our tool discovers alternative configurations and generate corresponding suggestions in practical time.

Methodology While assessing the effectiveness of the bot's responses to answer the first research question, we also measure the time taken by the bot in finding alternative configurations, generating corresponding recommendations in NL, and updating the domain models with the changes accepted by a modeller. To measure the time taken by the bot, we use three automated timers—one for measuring the time taken by the bot in finding the alternative configurations T_{AC}, a second one for measuring the time taken by

[13] https://en.wikipedia.org/wiki/Precision_and_recall.

[14] https://en.wikipedia.org/wiki/Precision_and_recall.

[15] https://en.wikipedia.org/wiki/F-score.

the bot in generating the suggestions in NL T_{SG}, and a third one for measuring the time taken by the bot in updating the domain models T_{DM}. After discovering an alternative configuration in the *Recommendation Component*, an intent is triggered to generate actions in the *Query Answering Component* so that corresponding suggestions can be presented to a modeller. When the modeller accepts the suggestions, our bot updates the domain model with the accepted suggestions. Therefore, the T_{AC} starts just after a modeller action is performed and ends when alternative configurations are discovered. The T_{SG} starts just after discovering an alternative configuration and ends when suggestions are presented to modellers in NL. Finally, the T_{DM} starts just after a modeller accepts a suggestion and ends when an extracted domain model is updated with the accepted suggestion. In addition, we measure the model size, i.e., the total number of elements present in an extracted domain model.

RQ3 (Usability): What are the benefits and limitations of our tool in supporting bot-modeller interactions?

Motivation: Multiple equivalent domain model solutions may exist for the same problem description. Therefore, an automated approach for generating a domain model from a given problem description should also consider modelling a scenario using different modelling decisions. Our bot aims to facilitate bot-modeller interactions so that an automatically generated domain model can quickly adapt to the modelling decisions which are preferred by a modeller (if they are not already included in the extracted model). For example, a scenario is modelled using the enumeration solution in an extracted domain model. However, a modeller may want to switch to the generalization solution to allow more flexibility for additional subtypes and for subtypes with different characteristics. Furthermore, the usefulness of our proposed tool ultimately depends on whether modellers find the tool helpful in real settings. Therefore, this research questions aims to assess the benefits and limitation of our tool in supporting bot-modeller interactions.

Methodology: We conduct a pilot user study where we hold semi-structured interviews with experienced modellers. The goal of this pilot user study is to assess the current benefits and limitations of our tool in facilitating bot-modeller interactions. Also, the results of this study may help us in designing a large-scale user study in the future more efficiently. In this study, we perform convenience sampling to recruit participants. At a high level, the study is composed of five parts. The first part (5 minutes) represents a preliminary survey which focuses on information about the modellers, with demographic questions about their general modelling experience and their experience in domain modelling. The second part (10 minutes) asks modellers to create a domain model manually for a given problem description and measures the time taken by them during this activity. This part also measures the accuracy of manually constructed domain models with respect to ground truth domain models. The third part (10 minutes) demonstrates how our tool enables bot-modeller interactions. The fourth part (15 minutes) asks modellers to use our tool for first extracting a domain model automatically for another problem description which is similar to the problem description used in the second part in terms of model size and complexity. In addition, this part measures the accuracy of automatically generated domain models with respect to corresponding ground truth domain models. Next, modellers interact with our tool to update the domain models so that their preferred modelling decisions can be included in the extracted domain model (if not included already). This part also measures the time taken by a modeller in confirming that the final state of the domain model is complete and includes their preferred modelling decisions. Finally, the fifth part conducts a semi-structured interview with the modellers to assess the usefulness and relevance of the bot's suggestions during bot-modeller interaction.

6 Results and discussion

In this section, we present the results of our evaluation with respect to our three research questions. In addition, we discuss the implications of evaluation results in relation to first and second research questions (RQ1 and RQ2) and the lessons learned from our pilot user study (RQ3). Our experiment data is available on the GitHub page[16].

6.1 Effectiveness (RQ1)

Results: First, we evaluate the accuracy of extracted domain models for each problem description in the PD_{eval} set [38, 40]. To calculate the accuracy of extracted domain models for the given problem descriptions, first, we generate the model extraction results using our bot for each problem description. Second, we consider these generated results to calculate the total retrieved concepts (classes, attributes, relationships, and cardinalities), i.e., TP + FP. Third, we consider the possible variations in the ground truth domain models and calculate the total retrieved relevant concepts from the extracted domain models, i.e, TP. In addition, we calculate the total relevant concepts, i.e., TP + FN, based on the selected variations in their corresponding ground truth domain models. Finally, we use the total relevant concepts, the total retrieved concepts, and the total retrieved relevant concepts to calculate the precision and recall scores for each problem description. The average precision and recall scores of extracted models using our proposed approach are 90% and 77%, respectively [38]. We follow the same recipe to calculate the precision and

[16] https://rijul5.github.io/Domain-Modelling-Bot/.

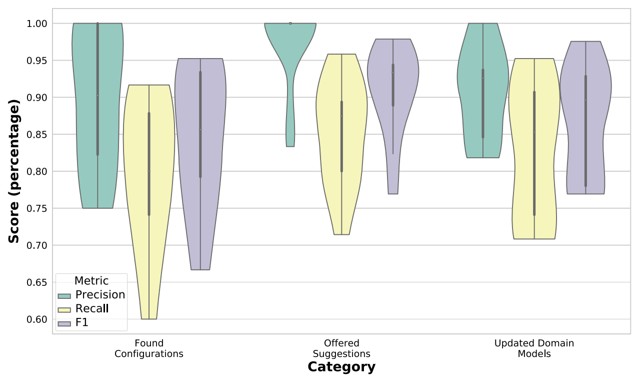


Fig. 6 Evaluation for Effectiveness (RQ1)

recall scores of manually constructed domain models while evaluating Research Question 3.

Finally, we initiate the bot-modeller interactions and evaluate the bot's responses. As shown in Figure 6, we use a violin plot which is a combination of a box plot and a kernel density plot to show the distribution of precision, recall, and F1 scores for *Found Configurations*, *Offered Suggestions*, and *Updated Domain Models* categories. The median precision score achieved by the *Found Configurations*, *Offered Suggestions*, and *Updated Domain models* categories is 90%, 100%, and 93%, respectively. The median recall score achieved by the *Found Configurations*, *Offered Suggestions*, and *Updated Domain models* categories is 80%, 87%, and 85%, respectively. Finally, the median F1 score achieved by the *Found Configurations*, *Offered Suggestions*, and *Updated Domain models* categories is 86%, 91%, and 90%, respectively. The *TP*, *FP*, *FN*, *Precision*, *Recall*, and *F1* scores of each category for the corresponding problem description are shown in Table 5.

Implications: Following our intuition, the evaluation results (see Figure 6) show that the median precision, recall, and F1 scores for *Offered Suggestions* and *Updated Domain Models* are higher than those of the *Found Configurations* category. One challenge in finding alternative configurations is that the discovery process for these configurations depends on the model extraction results (*Descriptive*, *Predictive*, and *Decision and Analysis Components*). Though we only consider the decision points from the ground truth domain models which are also present in the extracted domain models, the extraction of configurations along with the corresponding slices can still be inefficient due to incorrect or missing results from the model extraction process. Therefore, the discovery of configurations in the case of incorrect or incomplete model extraction results is challenging. Furthermore, we observe that *Found Configurations*, *Offered Suggestions*, and *Updated Domain Models* categories share a causal relationship. An incorrect or missing configuration which may be due to less accurate model extraction results can lead to low precision and recall scores in all the categories. However, if an alternative configuration is found then offering these suggestions and updating the domain model based on accepted changes is direct as they depend on the integration of the *Recommendation Component* with the *Query Answering* and *Communication Components*.

Moreover, we observe that the median precision, recall, and F1 scores for *Updated Domain Models* are higher than the *Found Configurations* category but lower than the *Offered Suggestions* category. This could be attributed to the conflicts which may arise when an alternative configuration (modelling decision) needs to be included in an extracted domain model. The offered configuration should be compatible with other existing configurations (which do not require any change). In many cases, conflicts may arise while switching to new configurations which can lead to inaccurate domain models. On the other side, presenting suggestions in NL to modellers is straightforward and easy. We further observe that the *Found Configurations* category has a lower recall score than precision score due to higher number of FN (relevant configurations missing in the candidate list of alternative configurations) than FP (non-relevant configurations in the candidate list of alternative configurations). This could be attributed to the decision threshold values which place a strict qualification while finding the possible configurations from the *Ranked Model Clusters* (ranked clusters with probability scores higher than decision threshold values). With this strict qualification, FP are reduced and FN are increased. Due to causal relationships, this behaviour is propagated to the *Offered Suggestions* and *Updated Domain Models* categories to some extent. In the future, we plan to investigate this further by fine-tuning the decision threshold values with a large set of problem descriptions.

6.2 Performance (RQ2)

Results: As shown in Table 5, we present the total number of domain model elements (Model Size in first row) present in each problem description and the time taken by the bot in finding configurations, offering suggestions, and in updating domain models for each problem description. The maximum number of elements (63) such as classes and relationships are present in the ninth problem description with 171 words while the minimum number of elements (11) are present in the third problem description with 36 words. We take the median of time taken by our bot for each problem description in three categories—Found Configurations, Offered Suggestion, and Updated Domain Models. We observe that our bot takes median time of 55.5, 11, and 19 milliseconds (ms) for a problem description (with median model size of 30.5 and median number of words of 106) to find configurations (T_{AC}), to offer suggestions to modellers (T_{SG}), and to update domain

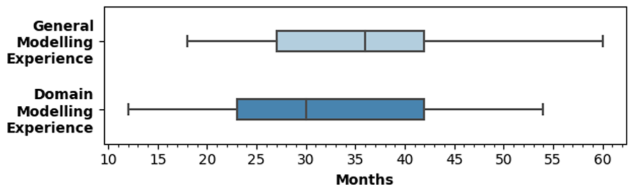


Fig. 7 Results of Preliminary Survey

models after the acceptance of offered suggestions by a modeller (T_{DM}), respectively.

Implications: We execute the experiments on a ThinkPad T470 laptop with Ubuntu (20.04.2 LTS) operating system, Intel i7 processor 2.70GHz, and 20Gb of RAM memory. We observe that there is very weak correlation between the model size and the time taken by the bot to find configurations (T_{AC}), offer suggestions (T_{SG}), and update domain models (T_{DM}). For example, the least amount of T_{AC} (49ms) is for the sixth problem description. This problem description has more elements than the third problem description which represents the smallest model size (11 elements) but still takes more time (62ms) than the sixth problem description. Therefore, the results from the second research question imply that the model size does not or very minimally effect the time performance of our bot during bot-modeller interactions. Furthermore, our bot finds alternative configurations, suggests these configurations to modellers in NL, and updates an extracted domain model in practical time, i.e., median time of 55.5ms, 11ms, and 19ms respectively.

6.3 Usability (RQ3)

Results: We invite 18 modellers for our exploratory study. Among these 18 modellers, 11 participate in the end (6 industry professionals and 5 graduate students). The results of the preliminary survey (first part of our study) are shown in Figure 7. We observe in our study that the participants have general modelling experience ranging from 18 months to 60 months (median of 36 months, i.e., 3 years). In addition, the participants have domain modelling experience ranging from 12 months to 54 months (median of 30 months, i.e., 2.5 years).

Furthermore, we obtain the results from the second and fourth parts of our study as shown in Table 6. Row (1) shows the time taken by all participants for manually constructing the domain models for the first problem description. The average time taken by participants for constructing domain models manually is 9.01 minutes (mins). Row (2) shows the accuracy of these constructed domain models with respect to corresponding ground truth domain models. The average accuracy of manually constructed domain models is 98.36%. Row (3) indicates the time (0.6 mins) taken by our bot to extract a domain model from the second problem description automatically. The first and second problem descriptions are similar in model size and complexity with approximately 35 model elements such as classes and relationships. Row (4) shows the accuracy of the extracted domain model is 87% with respect to the ground truth domain model.

During the fourth part of our study, participants interact with the bot to include their preferred modelling decisions in the extracted domain model (if not included already). The time and accuracy of extracted domain models are the same for every participant as these models are extracted using an automated approach which is based on rule-based NLP heuristics and ML models. Though these values are the same, we still illustrate them in Table 6 to analyze the total time saved and the improvement in accuracy of the extracted domain model during bot-modeller interaction. Row (5) indicates the bot-modeller interaction time period. This time period starts when an automatically extracted domain model is presented to a participant for the first time and ends when the participant is completely satisfied with the overall domain model, i.e., the final state of the extracted domain model which is obtained after the bot has pro-actively updated the original model based on accepted suggestions. The accuracy scores of the final state of these models is shown in Row (6). Finally, Row (7) provides the overall time taken for automated and interactive domain modelling using our bot, i.e., Row (3) + Row (5). The average total time for automated and interactive domain modelling is 2.28 mins.

In the interview, the participants evaluate the bot-modeller interactions facilitated by our tool and provide ratings for two aspects—usefulness and relevance, on a scale of 5 (1:low, 5:high). The relevance aspect evaluates if the bot's suggestions in response to modeller actions (changes made by a participant in an extracted domain model) are in line with the alternative configuration (modelling decision) that a participant wants to include in the extracted domain model. On the other side, the usefulness aspect evaluates if the bot's response to the acceptance of offered suggestions by a participant can modify the original modelling decision in an extracted domain model correctly and completely. Our tool achieves an average of 4.6 for the relevance aspect and 4.2 for the usefulness aspect.

Lessons Learned: First, our pilot user study identifies some areas of improvement for the future large-scale study design. For example, we can split participants into three groups - one control group (CG) and two study groups (SG1 and SG2). CG performs domain modelling for a given problem description (P1) without using our bot. SG1 performs manual domain modelling for P1 without using our bot and then performs domain modelling using our bot for a second problem description (P2). On the other side, SG2 performs domain modelling for P1 using our bot and then manually extracts domain models for P2. Second, our study shows that it is still possible for participants to miss out on some con-

Table 5 Evaluation for Effectiveness and Performance (RQ1 and RQ2)

Category	Metric	Domain Problem Descriptions (D)											
		D1	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**
	Model Size	51	23	11	30	31	39	19	43	63	25	33	17
Found Configurations	TP	6	5	3	6	7	7	4	13	22	8	10	4
	FP	2	1	1	1	1	0	1	1	1	0	0	0
	FN	3	2	2	2	2	1	1	2	2	1	1	1
	Precision(%)	75	83	75	86	87	100	80	93	96	100	100	100
	Recall(%)	67	71	60	75	78	87	80	87	92	89	91	80
	F1(%)	71	77	67	80	82	93	80	90	94	94	95	89
	T_{AC} (ms)	63	58	62	50	60	49	52	65	69	53	50	52
Offered Suggestions	TP	7	5	4	7	8	7	4	13	23	8	10	4
	FP	1	1	0	0	0	0	0	1	0	0	0	0
	FN	2	2	1	1	1	1	1	2	1	1	1	1
	Precision(%)	87	83	100	100	100	100	100	93	100	100	100	100
	Recall(%)	78	71	80	87	89	87	80	87	96	89	91	80
	F1(%)	82	77	89	93	94	93	89	90	98	94	95	89
	T_{SG} (ms)	8	12	12	11	7	8	5	11	7	11	12	11
Updated Domain Models	TP	17	13	5	9	9	13	5	20	26	13	12	7
	FP	3	1	1	2	1	1	1	0	1	1	1	0
	FN	7	2	2	3	1	1	2	1	5	1	4	1
	Precision(%)	85	93	83	82	90	93	83	100	96	93	92	100
	Recall(%)	71	87	71	75	90	93	71	95	84	93	75	87
	F1(%)	77	90	77	78	90	93	77	98	90	93	83	93
	T_{DM} (ms)	15	17	15	19	23	15	16	24	23	19	20	20

Table 6 Evaluation for Usability (RQ3)

Metric T: time (minutes), AC: accuracy (%), TT: total time (minutes)	**Participants (P)**										
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**
T: Manual Domain Modelling	8.5	10.1	8.3	11.2	9.1	9.0	7.9	10.3	8.9	7.6	8.3
AC: Manual Domain Model	97	100	94	98	100	98	100	100	99	100	96
T: Automatic Model Extraction	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
AC: Automatic Domain Model (Initial)	87	87	87	87	87	87	87	87	87	87	87
T: Bot-Modeller Interactions	1.8	1.1	1.7	1.1	2.3	1.5	2.5	1.5	2.1	1.1	1.8
AC: Automatic Domain Model (Final)	100	98	100	96	100	99	98	97	100	100	100
TT: Domain Modelling using Bot	2.4	1.7	2.3	1.7	2.9	2.1	3.1	2.1	2.7	1.7	2.4

cepts when they are constructing domain models or reviewing the extracted domain models for their preciseness and completeness due to time constraints. For example, Participant 1, missed some concepts initially. However, when additional time is given to the participant for reviewing the manually constructed domain model, then the participant was able to identify the missing concepts. Therefore, we may need to keep the manual domain modelling activity free of a time constraint so that the participants can come up with the best possible configuration of the domain model based on their knowledge and experience. These manually created configurations can then be compared with the ground truth domain models to compute their accuracy. Moreover, these accuracy results of manually constructed domain models may represent a better baseline for our tool while evaluating the automatically created domain models using our proposed bot.

Third, we observe from our pilot study that the average total time for automated and interactive domain modelling using our bot is 2.28 mins. Therefore, our bot may reduce modeller's time on an average by 74.69% for a problem description of similar size and complexity (P1 and P2). Furthermore, the accuracy of the automatically extracted domain model is 87% with respect to the ground truth domain model and the average accuracy of manually constructed domain models is 98.36%. Though the average accuracy of automatically extracted domain models is lower than the average accuracy of manually constructed domain models, the time taken by our proposed bot to extract the domain model with reasonable accuracy and then to adapt the extracted domain model based on the configurations preferred by a modeller, is much shorter (an average time of 2.28 mins). Therefore, we argue that our proposed bot can assist modellers significantly in quickly prototyping the domain models for the problem descriptions which are similar to P1 and P2 problem descriptions in terms of model size and complexity.

Fourth, participants agree that our bot provides suggestions and updates the domain model in real-time. Participant 9 commented that the direct integration of the model extraction process into the recommendations generation process shows plenty promise where the bot offers alternative configurations (suggestions) on the fly during bot-modeller interactions. Fifth, participants agree that our bot improves the awareness of different modelling decisions (alternative configurations) by providing messages in NL. Participant 7 commented that the offered suggestions are direct and intuitive for learning the impact of accepting suggestions in an extracted domain model. Finally, participants provide some suggestions to improve our tool in supporting bot-modeller interactions. For example, the scope of an offered suggestion, i.e., the concepts such as classes and relationships which will be impacted under this suggestion can be presented graphically in an extracted domain model along with the current textual NL messages. Also, after updating an extracted domain model on the acceptance of an offered suggestion, the bot may support an undo feature so that a modeller can switch back to the previous modelling decision (configuration) if required.

7 Threats to validity

In this section, we discuss the validity factors most pertinent to the evaluation of our bot for its effectiveness, performance, and usability.

A. Construct Validity: With regard to construct validity, we note that our evaluation considers a small number of problem descriptions in the validation set PD_{eval}. In the future, we plan to perform an evaluation with a higher number of domain problem descriptions. In addition, we construct the ground truth which represents the manually created domain model solutions as well as the identified decision points (with two or more configurations). The construction of this ground truth data may include some errors. However, we mitigate this threat by a second author reviewing the ground truth data created by one author so that the decisions can be cross-checked and confirmed. In addition, if there are any conflicts then both authors discuss the conflicts and resolve them by making the required changes to the ground truth data.

B. External Validity: Threats to external validity have to do with the generalizability of our results to other problem descriptions. The problem descriptions in the validation set PD_{eval} may not represent a wide sample of domain problems in terms of problem complexity and model size. Though we ensure that these randomly selected problem descriptions exhibit at least one decision point (two or more configurations), these problem descriptions still cannot represent all possible domain modelling scenarios. To reduce this threat to some extent, we created a larger problem description with more than 300 model elements and evaluate the time performance. The average time taken by our bot to find configuration, present suggestions to modellers, and to update domain models is still under 55ms. In the future, we plan to perform an evaluation with a bigger and more diverse dataset.

C. Internal Validity: Threats to internal validity have to do with whether other plausible hypotheses could explain our pilot study results. We perform convenience sampling to recruit participants for our pilot study. Our bot shows promise based on the interview results in this study. However, the participants may be biased towards providing positive feedback to us due to social pressure. To combat this, we explained to all participants that their direct and honest feedback was what we needed to collect in order to improve our proposed bot. Nevertheless, the response may still have been biased towards the positive. For the future large-scale study, we plan to use other methods of recruitment, so that the participants are more representative of the general population of modellers. In addition, threats to internal validity also have to do with the use of only one study group to evaluate our tool to answer RQ3. During the second part of our study, participants perform domain modelling manually for the first problem description. It may be possible for participants to learn about applying modelling patterns such as the Player-Role pattern more easily when using our tool to perform domain modelling for a second problem description. The similarity of the first and second problem descriptions in terms of model size and complexity also emphasizes this threat. Therefore, one possible solution to combat this threat is to form two different study groups to perform domain modelling activities (one manually first and second using our tool; the other using our tool first and second manually) in addition to a control group. We plan to use this approach for our large-scale study in the future.

8 Conclusions and future work

Domain modelling requires time, modelling skills, and knowledge. The domain models generated automatically from the existing approaches are not accurate enough to be used directly or with small changes in software development. In addition, existing approaches do not provide interactive interfaces to enable modellers to update the extracted models and to assist them in gaining insights into the modelling decisions taken by the extractor. To overcome these limitations and challenges, we introduce a domain modelling bot called DoMoBOT and explain its architecture [38,40]. The proposed bot generates the domain models for given problem descriptions in NL and enables modellers to interact with the bot for updating the extracted domain models quickly.

In this paper, we add *Recommendation* and *Query Answering Components* to the architecture of DoMoBOT for facilitating bot-modeller interactions. We propose an algorithm to discover alternative configurations during bot-modeller interactions. Our bot uses this algorithm to find alternative configurations and then present these configurations in the form of suggestions to modellers. Our bot also updates the domain model in response to the acceptance of these suggestions by a modeller. Furthermore, we implement the new components of our architecture and our proposed algorithm in the form of a web-based tool and present our tool in this paper. Finally, we perform the evaluation of our bot for its effectiveness, performance, and usability. Our bot achieves the median F1 scores of 86%, 91%, and 90% in the *Found Configurations*, *Offered Suggestions*, and *Updated Domain Models* categories, respectively. In addition, we show that our bot takes median time of 55.5, 11, and 19 ms for a problem description (with median model size of 30.5 and median number of words of 106) to find configurations, to offer suggestions to modellers, and to update domain models, respectively. Finally, we present the lessons learned from our pilot user study.

In future work, first, we plan to perform a large-scale user study with modellers so that the relevance and usefulness of our tool in facilitating the bot-modeller interactions can be assessed more thoroughly. Second, we plan to evaluate our approach with a higher number of domain problem descriptions and also with those problem descriptions which are very long with model size beyond 100 elements. Third, we aim to improve the model extraction results by enhancing the qualifiers in *Descriptive*, *Predictive*, and *Analysis and Decision Making Components*. Fourth, we plan to use the bot-modeller interaction story, i.e., a series of queries and responses between our bot and a modeller to improve the quality of suggestions which are provided by our bot. Fifth, we plan to present the scope of offered suggestions graphically to make modellers aware of the impact of offered suggestions when accepted. Finally, we plan to enhance our tool for supporting our ultimate goal which is to assist instructors or experienced modellers in quickly building the domain models, tracing the modelling decisions in an extracted domain model, and to participate in bot-modeller interactions so that their preferred modelling decisions can be included automatically in the extracted domain model.

References

1. Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.C., Rummler, A., Sousa, A.: A model-driven traceability framework for software product lines. Softw. Syst. Model. **9**(4), 427–451 (2010). https://doi.org/10.1007/s10270-009-0120-9
2. Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F.: Extracting domain models from natural-language requirements: Approach and industrial evaluation. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, p. 250-260. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976767.2976769
3. Ben Charrada, E., Caspar, D., Jeanneret, C., Glinz, M.: Towards a benchmark for traceability. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, IWPSE-EVOL '11, p. 21–30. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2024445.2024451
4. Bencomo, N., Garcia Paucar, L.H.: RaM: causally-connected and requirements-aware runtime models using Bayesian learning. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 216–226 (2019). https://doi.org/10.1109/MODELS.2019.00005
5. Bousse, E., Mayerhofer, T., Combemale, B., Baudry, B.: A generative approach to define rich domain-specific trace metamodels. In: 11th European Conference on Modelling Foundations and Applications (ECMFA). L'Aquila, Italy (2015). https://hal.inria.fr/hal-01154225
6. Burgueño, L., Clarisó, R., Gérard, S., Li, S., Cabot, J.: An nlp-based architecture for the autocompletion of partial domain models. In: La Rosa, M., Sadiq, S., Teniente, E. (eds.) Advanced Information

Systems Engineering, pp. 91–106. Springer International Publishing, Cham (2021)

7. Burgueño, L., Cabot, J., Gérard, S.: An LSTM-based neural network architecture for model transformations. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 294–299 (2019). https://doi.org/10.1109/MODELS.2019.00013
8. Cerqueira, T.G.O., Ramalho, F., Marinho, L.B.: A content-based approach for recommending UML sequence diagrams. In: Gou, J. (ed.) The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016, Redwood City, San Francisco Bay, USA, July 1–3, 2016, pp. 644–649. KSI Research Inc. and Knowledge Systems Institute Graduate School (2016). https://doi.org/10.18293/SEKE2016-147
9. Chen, X., Hosking, J., Grundy, J.: A combination approach for enhancing automated traceability: (nier track). In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 912–915 (2011). https://doi.org/10.1145/1985793.1985943
10. Chen, X., Hosking, J., Grundy, J., Amor, R.: Development of robust traceability benchmarks. In: 2013 22nd Australian Software Engineering Conference, pp. 145–154 (2013). https://doi.org/10.1109/ASWEC.2013.26
11. Cleland-Huang, J., Czauderna, A., Dekhtyar, A., Gotel, O., Hayes, J.H., Keenan, E., Leach, G., Maletic, J., Poshyvanyk, D., Shin, Y., Zisman, A., Antoniol, G., Berenbach, B., Egyed, A., Maeder, P.: Grand challenges, benchmarks, and tracelab: Developing infrastructure for the software traceability research community. In: Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE '11, p. 17-23. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1987856.1987861
12. Cleland-Huang, J., Hayes, J.H., Domel, J.M.: Model-based traceability. In: 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp. 6–10 (2009). https://doi.org/10.1109/TEFSE.2009.5069575
13. Elkamel, A., Gzara, M., Ben-Abdallah, H.: An UML class recommender system for software design. In: 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), pp. 1–8 (2016). https://doi.org/10.1109/AICCSA.2016.7945659
14. Fawcett, T.: An introduction to roc analysis. Pattern Recogn. Lett. **27**(8), 861–874 (2006). https://doi.org/10.1016/j.patrec.2005.10.010
15. Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., Mäder, P.: Traceability Fundamentals, pp. 3–22. Springer London, London (2012). https://doi.org/10.1007/978-1-4471-2239-5_1
16. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
17. Houmb, S.H., Islam, S., Knauss, E., Jürjens, J., Schneider, K.: Eliciting security requirements and tracing them to design: an integration of common criteria, heuristics, and UMLsec. Requirements Eng. **15**(1), 63–93 (2010). https://doi.org/10.1007/s00766-009-0093-9
18. Hübner, P., Paech, B.: Interaction-based creation and maintenance of continuously usable trace links between requirements and source code. Empir. Softw. Eng. **25**(5), 4350–4377 (2020). https://doi.org/10.1007/s10664-020-09831-w
19. Ibrahim, M., Ahmad, R.: Class diagram extraction from textual requirements using natural language processing (NLP) techniques. In: 2010 Second International Conference on Computer Research and Development, pp. 200–204 (2010). https://doi.org/10.1109/ICCRD.2010.71
20. Jyothilakshmi, M.S., Samuel, P.: Domain ontology based class diagram generation from functional requirements. In: 2012 12th International Conference on Intelligent Systems Design and Applications (ISDA), pp. 380–385 (2012). https://doi.org/10.1109/ISDA.2012.6416568
21. Kassab, M., Ormandjieva, O., Daneva, M.: A traceability metamodel for change management of non-functional requirements. In: 2008 Sixth International Conference on Software Engineering Research, Management and Applications, pp. 245–254 (2008). https://doi.org/10.1109/SERA.2008.37
22. Landhäußer, M., Körner, S.J., Tichy, W.F.: From requirements to UML models and back: how automatic processing of text can support requirements engineering. Software Qual. J. (2014). https://doi.org/10.1007/s11219-013-9210-6
23. Lucrédio, D., Fortes, M., R.P., Whittle, J.: MOOGLE: a metamodel-based model search engine. Softw. Syst. Model. **11**(2), 183–208 (2012)
24. Mader, P., Gotel, O., Philippow, I.: Getting back to basics: promoting the use of a traceability information model in practice. In: 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp. 21–25 (2009). https://doi.org/10.1109/TEFSE.2009.5069578
25. Maro, S., Steghofer, J., Knauss, E., Horkof, J., Kasauli, R., Wohlrab, R., Korsgaard, J., Wartenberg, F., Strom, N., Alexandersson, R.: Managing traceability information models: not such a simple task after all? IEEE Software. https://doi.org/10.1109/MS.2020.3020651
26. Miller, G.A.: Wordnet: A lexical database for english. Commun. ACM **38**(11), 39–41 (1995). https://doi.org/10.1145/219717.219748
27. Miller, G.A.: WordNet: An electronic lexical database. MIT press (1998). https://wordnet.princeton.edu/
28. Montes, A., Pacheco, H., Estrada, H., Pastor, O.: Conceptual model generation from requirements model: A natural language processing approach. In: International Conference on Application of Natural Language to Information Systems, pp. 325–326. Springer (2008)
29. Mora Segura, A., Pescador, A., de Lara, J., Wimmer, M.: An extensible meta-modelling assistant. In: 2016 IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC), pp. 1–10 (2016). https://doi.org/10.1109/EDOC.2016.7579377
30. Mussbacher, G., Combemale, B., Kienzle, J., Abrahão, S., Ali, H., Bencomo, N., Búr, M., Burgueño, L., Engels, G., Jeanjean, P., et al.: Opportunities in intelligent modeling assistance. Softw. Syst. Model. **19**(5), 1045–1053 (2020). https://doi.org/10.1007/s10270-020-00814-5
31. Pennington, J., Socher, R., Manning, C.: GloVe: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543. Association for Computational Linguistics, Doha, Qatar (2014). 10.3115/v1/D14-1162. https://aclanthology.org/D14-1162
32. Pinheiro, F., Goguen, J.: An object-oriented tool for tracing requirements. In: Proceedings of the Second International Conference on Requirements Engineering, pp. 219– (1996). https://doi.org/10.1109/ICRE.1996.491449
33. Pérez-Soler, S., Guerra, E., de Lara, J.: Collaborative modeling and group decision making using chatbots in social networks. IEEE Softw. **35**(6), 48–54 (2018). https://doi.org/10.1109/MS.2018.290101511
34. Pérez-Soler, S., Guerra, E., de Lara, J., Jurado, F.: The rise of the (modelling) bots: towards assisted modelling via social networks. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 723–728 (2017). https://doi.org/10.1109/ASE.2017.8115683
35. Robeer, M., Lucassen, G., van der Werf, J.M.E.M., Dalpiaz, F., Brinkkemper, S.: Automated extraction of conceptual models from user stories via NLP. In: 2016 IEEE 24th International Require-

ments Engineering Conference (RE), pp. 196–205 (2016). https://doi.org/10.1109/RE.2016.40
36. Saini, R., Mussbacher, G., Guo, J.L., Kienzle, J.: A neural network based approach to domain modelling relationships and patterns recognition. In: 2020 IEEE Tenth International Model-Driven Requirements Engineering (MoDRE), pp. 78–82 (2020). https://doi.org/10.1109/MoDRE51215.2020.00016
37. Saini, R., Mussbacher, G., Guo, J.L., Kienzle, J.: Towards queryable and traceable domain models. In: 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 334–339 (2020). https://doi.org/10.1109/RE48521.2020.00044
38. Saini, R., Mussbacher, G., Guo, J.L., Kienzle, J.: Automated traceability for domain modelling decisions empowered by artificial intelligence. In: 2021 IEEE 29th International Requirements Engineering Conference (RE) (to be published) (2021)
39. Saini, R., Mussbacher, G., Guo, J.L.C., Kienzle, J.: Teaching modelling literacy: An artificial intelligence approach. In: MODELS 2019 Companion, pp. 714–719 (2019). https://doi.org/10.1109/MODELS-C.2019.00108
40. Saini, R., Mussbacher, G., Guo, J.L.C., Kienzle, J.: 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, MODELS '20. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3417990.3421385
41. Salih Dawood, O., Sahraoui, A.E.K.: From Requirements Engineering to UML using Natural Language Processing—Survey Study. Eur. J. Ind. Eng. **2**, 44–50 (2017)
42. Sardinha, A., Yu, Y., Niu, N., Rashid, A.: EA-Tracer: identifying traceability links between code aspects and early aspects. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, pp. 1035–1042. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2245276.2231938
43. Savary-Leblanc, M.: Improving MBSE tools UX with AI-empowered software assistants. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 648–652 (2019). https://doi.org/10.1109/MODELS-C.2019.00099
44. Schisterman, E.F., Perkins, N.J., Liu, A., Bondell, H.: Optimal cut-point and its corresponding youden index to discriminate individuals using pooled blood samples. Epidemiology pp. 73–81 (2005). https://doi.org/10.1097/01.ede.0000147512.81966.ba
45. Schlutter, A., Vogelsang, A.: Knowledge representation of requirements documents using natural language processing. In: K. Schmid, P. Spoletini, E.B. Charrada, Y. Chisik, F. Dalpiaz, A. Ferrari, P. Forbrig, X. Franch, M. Kirikova, N.H. Madhavji, C. Palomares, J. Ralyté, M. Sabetzadeh, P. Sawyer, D. van der Linden, A. Zamansky (eds.) Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), Utrecht, The Netherlands, March 19, 2018, *CEUR Workshop Proceedings*, vol. 2075. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2075/NLP4RE_paper9.pdf
46. Schöttle, M., Thimmegowda, N., Alam, O., Kienzle, J., Mussbacher, G.: Feature modelling and traceability for concern-driven software development with touchCORE. In: Companion Proceedings of the 14th International Conference on Modularity, MODULARITY Companion 2015, p. 11-14. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2735386.2735922
47. Sharma, R., Srivastava, P.K., Biswas, K.K.: From natural language requirements to UML class diagrams. In: 2015 IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE), pp. 1–8 (2015). https://doi.org/10.1109/AIRE.2015.7337625
48. Stirewalt, R.E.K., Deng, M., Cheng, B.H.C.: UML formalization is a traceability problem. In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE '05, p. 31-36. Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1107656.1107664
49. Youden, W.J.: Index for rating diagnostic tests. Cancer **3**(1), 32–35 (1950)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Rijul Saini is a Ph.D. candidate at McGill University in Montreal, Canada. His research interests include model-driven requirements engineering, modelling assistance using machine learning and natural language processing, and collaborative modelling. He is a PC member of the MDE Intelligence Workshop at the MODELS Conference. Contact him at rijul.saini@mail.mcgill.ca.

Gunter Mussbacher is an Associate Professor in the Department of Electrical and Computer Engineering at McGill University, with 150+ publications related to model-driven requirements engineering, self-adaptive languages, next-generation reuse frameworks, sustainability, and human values. He currently holds an INRIA International Chair. He has co-edited all versions of the User Requirements Notation (URN), an international requirements engineering standard published by the International Telecommunication Union as ITU Recommendation Z.151, and is currently Associate Rapporteur responsible for URN at ITU. He was Program Co-Chair for SAM'14, Finance Chair for RE'15, Conference Chair for ICSE 2019, and is currently Program Co-Chair for RE'22. He co-founded the Model-Driven Requirements Engineering (MoDRE) workshop series at RE, is a frequent organizer of workshops, and is a regular PC member of key conferences in his field.

Jin L. C. Guo is an assistant professor in the School of Computer Science at McGill University. She received her Ph.D. in Computer Science and Engineering from the University of Notre Dame. She is particularly interested in the intersection between Software Engineering, Human-Computer Interaction, and Artificial Intelligence. Her recent research focuses on software traceability, OSS usability, and software documentation.

Jörg Kienzle is an associate professor at the School of Computer Science at McGill University in Montreal, Canada. He holds a Ph.D. and engineering diploma from the Swiss Federal Institute of Technology in Lausanne (EPFL). His current research interests include model-driven engineering, concern-oriented software development, reuse of models, software development methods in general, aspect-orientation, distributed systems and fault tolerance. Contact him at Joerg.Kienzle@mcgill.ca.