Ontology-Based Models of Chatbots for Populating Knowledge Graphs

Petko Rutesic[1](✉), Dennis Pfisterer[2], Stefan Fischer[2], and Heiko Paulheim[3]

[1] Baden-Wuerttemberg Cooperative State University, 68163 Mannheim, Germany
petko.rutesic@dhbw-mannheim.de
[2] Institute of Telematics, University of Luebeck, Luebeck, Germany
[3] University of Mannheim, Mannheim, Germany

Abstract. Knowledge graphs and graph databases are nowadays extensively used in various domains. However, manually creating knowledge graphs using existing ontology concepts presents significant challenges. On the other hand, chatbots are one of the most prominent technologies in the recent past. In this paper, we explore the idea of utilizing chatbots to facilitate the manual population of knowledge graphs. To implement these chatbots, we generate them based on other special knowledge graphs that serve as models of chatbots. These chatbot models are created using our modelling ontology (specially designed for this purpose) and ontologies from a specific domain. The proposed approach enables the manual population of knowledge graphs in a more convenient manner through the use of automatically generated conversational agents based on our chatbot models.

Keywords: modelling · ontology · chatbots · knowledge graphs

1 Introduction

Creating user-friendly interfaces to facilitate populating knowledge graphs manually is a very demanding task. In order to create a knowledge graph, it is necessary to have expertise not only in the domain of interest but also in the field of ontology engineering. We can illustrate this using an example of flight registration, where the end-users enter flight information manually in the knowledge graph and the ultimate output of the process would be a comprehensive knowledge graph representing all existing flights. To define a specific flight, it is necessary to first choose the appropriate ontology for flight description and then create an individual of the class representing flights. Following that, the user has to know how to define departure and arrival airports, which requires the knowledge of object and data properties that can be used to describe airports. What makes this task even more complex is the need to choose whether these airports can be described as blank nodes or not, or to choose specific ontology design patterns. Creating these ontologies (editing RDF graphs) using only text editors in any syntax for representing RDF graphs would be intimidating for the majority of users.

Tools for ontology engineering like Protégé, WebProtégé, TopBraid Composer and similar tools are frequently used for this purpose. Additionally, there are various approaches to modelling user interfaces for populating knowledge graphs. These user interfaces are of different kinds, from desktop and web applications to conversational agents (chatbots). Particularly, chatbots have seen great growth in popularity, especially in the last couple of years with the appearance of large language models and tools like ChatGPT that use deep learning models to generate correct humanlike responses. Chatbots are now integrated in various domains and have numerous applications, such as e-customer care services, e-commerce systems, the medical field, etc.

Our approach aims to empower end users to create knowledge graphs like the aforementioned flight knowledge graph in a simple manner. The chatbot should primarily ask simple questions like: "What is the flight number?" or "What is the flight destination?". To this end, we propose to divide population of knowledge graphs in two processes. The first process would be modelling and designing of conversational agents, and the second process is using those conversational agents by many end users to populate desired knowledge graphs. One potential group of end users who could benefit from our chatbots includes airline personnel (or flight operators) responsible for registering new flights and services related to those flights. The novelty of our idea lies in the automatic generation of chatbots from models that themselves are knowledge graphs. By adopting this approach, the design of our conversational agent models becomes the responsibility of ontology experts (chatbot model designers), while the end users of the chatbots do not necessarily need expert knowledge in ontologies. This way, the end user would be relieved from the burden of precisely knowing domain ontologies and the intricacies of ontology engineering.

The approach can be simply expressed through two functions. The first function is named the modelling function, which encompasses the process of modelling chatbot dialogues (conversations) while simultaneously defining the structure of the output knowledge graphs. The modelling function takes a set of domain ontologies and our *OBOP* ontology (Ontology for Ontology-based Ontology Population) as input parameters. The *OBOP* ontology is specifically designed for modelling purposes and can be accessed at http://purl.org/net/obop or in the GitHub repository[1].

$$f_{modelling}(DomainOntologies, OBOP) = ChatbotModel$$

The modelling process is the task for chatbot model designers (ontology experts), and the output of this process is a $ChatbotModel$, which is a knowledge graph defined using elements from *DomainOntologies* and our *OBOP* ontology. Currently, the modelling is done manually, but we have plans to design special GUI tools to support and automate this process in the future.

[1] https://github.com/ontosoft/logic-interface/blob/main/ontology/obop.owl.

The second function represents the process of data acquisition, i.e., knowledge graph population. This process is actually the use of chatbot to enter data.

$$f_{acquisition}(ChatbotModel, UserInteraction) = OutputKnowledgeGraph$$

The acquisition function takes a chatbot model created in the modeling process and user interaction (during which the data is entered) as input parameters. The output of the acquisition function is a desired knowledge graph, referred to as *OutputKnowledgeGraph*, which is defined only using elements from the domain ontologies. In Fig. 1, part of our approach that corresponds to the acquisition function is outlined. The *Chatbot generator* takes the *chatbot model* defined by elements of the OBOP ontology (colored in red) and domain ontologies (colored in black). The *output knowledge graph* is populated through the interaction of the end-user with the chatbot and contains only elements from domain ontologies.

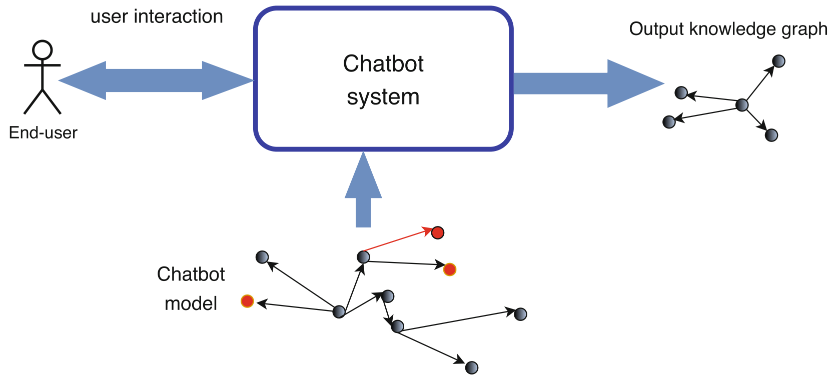

Fig. 1. A simplified representation of the acquisition function (Color figure online)

Since the main goal is to simplify the population of knowledge graphs using conversational systems (chatbots), we have explored the idea of modelling these chatbots also within knowledge graphs. To leverage the reasoning capabilities of OWL ontologies, it is reasonable for the chatbot models to be represented in the OWL DL ontology class. Therefore, the goal of our approach can be boiled down to the following set of requirements:

1. The chatbot conversation and the structure of the output knowledge graph are both specified within the same knowledge graph (e.g. RDF file), representing a model.
2. Models are defined using a dedicated ontology designed for this purpose, serving as a meta-model for generating our models.
3. The meta-model comprises elements capable of modelling main program control structures, i.e., sequential, selection (branching) and iteration control structures.

In the rest of the paper, we elaborate on the implementation of the proposed requirements. The rest of the paper is organized as follows: The next section presents a use case with flight registrations, used to illustrate our approach in subsequent sections. Then we describe the main elements of the OBOP ontology that model various aspects of chatbot conversations and showcase how the modeling function works in our use case. Each subsection focuses on specific workflows of the chatbot and explains how these workflows are modeled using entities from the OBOP ontology. Finally, the paper concludes with a section discussing our contributions.

2 Use Case

To demonstrate the applicability of our chatbot models, we introduce an example involving a chatbot designed to help create knowledge graphs for flights. Essentially, we look at process from the reverse perspective, beginning with an existing knowledge graph and assuming it to be the output knowledge graph of a chatbot. Then we demonstrate a model of this chatbot that generates this particular knowledge graph.

The knowledge graph that our chatbot has to construct is already presented in the examples of using the Ticket ontology[2]. The authors of the knowledge graph used GoodRelations ontology [6], Ticket ontology (compliant with GoodRelations) and DBpedia to describe the flight. A simplified graphical representation of the flight knowledge graph is shown in Fig. 2.

The flight knowledge graph contains data about a specific flight with the flight number LH1234, which is operated by Lufthansa airline. The flight information includes details about the departure and destination airports, as well as the types of tickets that can be booked for that flight. A flight ticket is defined as an instance of the class tio:Ticket of the Ticket ontology. In the graphical representation, individuals and blank nodes are depicted using circles, with blank nodes represented by empty circles. Ontology classes are illustrated by ovals, while literals are shown as rectangles.

During the interaction with the chatbot, the user must provide the flight number, departure and destination airports, as well as the departure and arrival times. The knowledge graphs (instances of the Ticket ontology) generated using our chatbot offer significant usefulness as they can be easily searched by customers looking to book flight tickets. The use of ontologies in the system allows for defining search operations using SPARQL queries. Moreover, customers have the option to employ conversational agents like KBot [2] to find suitable flight tickets using natural language understanding over linked data. In that case, semantic web technologies can be used directly to find suitable flights without resorting to web scraping methods as described in [12].

The primary objective of our chatbot use case is to simplify the process for end-users by posing straightforward questions, while the knowledge graph is

[2] http://purl.org/tio/ns.

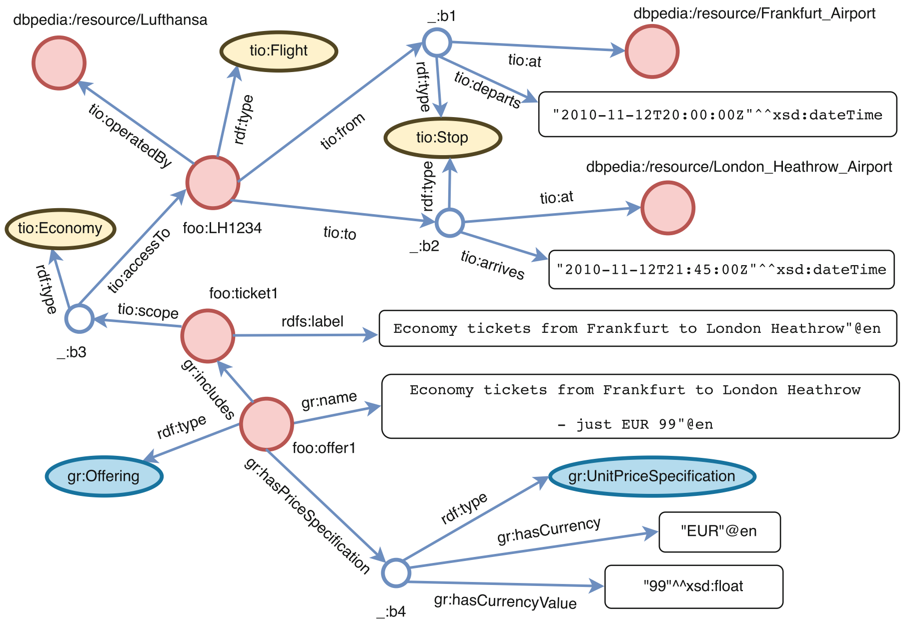

Fig. 2. A knowledge graph representing a simple flight description.

generated seamlessly in the background using answers to those questions. The chatbot must have the capability to generate this knowledge graph and also to accommodate its extensions, such as adding the definition of additional ticket types if needed. By doing so, the chatbot determines the structure of the output ontology without explicitly asking the user to specify blank nodes, instances of specific ontology classes and other details, thereby alleviating the burden of detailed ontology engineering. To achieve this level of sophistication, the chatbot model incorporates all the necessary details, and the key aspects of the model are outlined in the section dedicated to modelling architecture.

3 Related Work

One of the first known conversational agents is ALICE [13] which uses AIML (Artificial Intelligence Markup Language) which is basically XML to design conversations. The system uses a botmaster which monitors conversations to make them more appropriate. Unlike writing rules in an XML-based language, the rules in our approach are encoded in a knowledge graph based on our modeling ontology and can be stored in an RDF file. Historically, there have been various approaches to using semantic nets to generate chatbot. One of the first examples is OntBot [3] that transforms ontologies and knowledge into relational database and then uses relational database to generate chats. Another example of usage of model-driven conversational systems is presented in [9], wherein a specialized domain-specific language with components like intents, entities, actions and flows is employed to design the dialogue structure of task-oriented chatbots. The dialogue management used in our paper is also similar to the dialogue management used by the Rasa chatbot [5]. However, our approach differentiates itself from those approaches by aiming to represent all components, including intents, flows and actions through the use of knowledge graphs.

There have been many approaches to model user interfaces and web application logic using knowledge graphs. An approach described in [11] proposes a method of modelling HTML application structure and its logic using RDF graphs. In this paper, we deal with a similar question of modelling and generating chatbots used to engage in dialogs with users to acquire data. Furthermore, the acquired data is used to directly populate desired output knowledge graphs based on domain ontologies. The expressiveness of formal ontologies proves valuable in representing both the models of chatbots and the complex knowledge within the business logic of target ontologies. Chatbots generated using this approach could prove advantageous wherever the description of complex products and services using ontologies is required, as is the case in emerging business models like *Distributed Market Spaces* [10]. This approach complements the method described in [7], where the system utilizes SPARQL queries to identify appropriate complex products and services.

As stated in [1], conversational clients or chatbots are designed to be used either as task-oriented or open-ended dialog generators. In our approach, we develop a task-oriented chatbot responsible for collecting data (knowledge graph) based on particular ontologies. The actual task is described based on rules specified in the model, which is represented again as a knowledge graph defined using our OBOP ontology. Thus, our chatbot can also be classified as a rule-based chatbot. To enable human-like conversation, our task-based chatbot module is incorporated into an open-ended chatbot which is further described in the implementation section. The intention of our system is not to design a chatbot capable of convincing a human that (s)he is chatting with a human instead of a computer program. That intelligent behaviour depends on having good knowledge sets. Instead, we focus on the creation of chatbots that gather information in the form of knowledge graphs with individuals (instances) of the respective domain ontologies.

4 Model Architecture

Our meta-model (OBOP ontology) contains various structures that describe chatbot functionalities. The fundamental structure is designed to specify a simple data request. In response to this request, the user inputs a value, which is then validated and can be stored in the system as a data property, IRI, or for a similar purpose. This functionality corresponds to the process of entering data into form fields in GUI interfaces. To represent the insertion of these data values, the OBOP ontology, uses an instance of the class obop:SingleValueRequest. In cases where multiple data properties need to be entered as a group of questions, this is modeled using a conversation block. The generation of chatbot models, using constructions explained in the next sections, is the responsibility of chatbot model designers.

4.1 Conversation Block

A conversation block represents a segment of a conversation used to collect data values that are related to a specific entity. This part of the dialog is analogous to an HTML form in web applications. The chatbot poses a series of questions, and by providing answers to those questions, the system stores corresponding values. A simple model for entering the flight IRI is illustrated in Fig. 3. In the following figures, blue circles represent instances of classes of the OBOP ontology, while larger peach-colored circles (e.g., individual *flight_ model_ 1* in Fig. 3) depict instances from domain (target) ontologies. Yellow ovals represent classes from the OBOP ontology, and blue ovals represent classes from target ontologies. Object and data properties are denoted by directed lines with labels that specify the names of those properties.

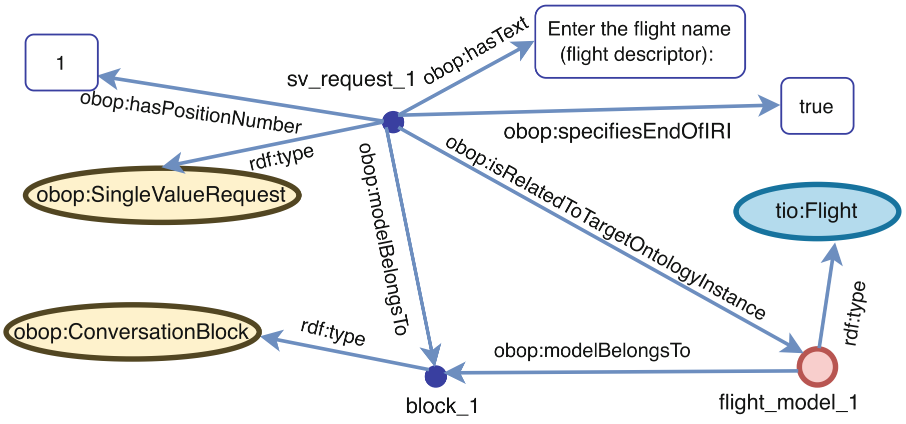

Fig. 3. Modelling a block of conversation

A segment of the model represented in Fig. 3 should initiate the following part of conversation:

```
chatbot: Please enter the flight name (flight descriptor):
user:    LH1234
```

The outcome of the previous conversation is adding of the following triple to the output knowledge graph:

```
foo:LH1234 a tio:Flight.
```

In Fig. 3 can be seen a conversation block named *block_ 1*, which has one instance of the class obop:SingleValueRequest called *sv_ request_ 1*. This represents a question for entering a flight name (a unique flight descriptor). *sv_ request_ 1* has the object property obop:specifiesEndOfIRI, which has a boolean value of true. This indicates that the provided answer to the question serves as the ending part of the IRI representing the flight instance. The model of the instance of tio:Flight class that will be generated in this process is represented by the *flight_ model_ 1* instance. During the conversation with the chatbot from this instance will be generated the instance with the name foo:LH1234. Should the inserted string need to be the value of a data property instead of being part of the IRI, this would be indicated by the *obop:containsDatatype* object property of the *sv_ request* instance which would specify the name of the wanted data property. The object property *obop:isRelatedToTargetOntologyInstance* indicates what instance will be transformed with the entered data. The data property *obop:hasPositionNumber* specifies that this question is the first one in this conversational block.

4.2 Branching Control Structure

To enable chatbot users to make decisions and select different paths of execution, the OBOP ontology has mechanisms for modeling branching control structures. The class *obop:Branching* represents a conditional structure. A simple branching structure in our example is explained in the case of selecting an airline that operates the given flight. The chatbot user is therefore prompted to choose only one airline company from the presented options. One possible conversation is presented in the following listing:

```
chatbot: Enter the airline that operates your flight.
         Choose one of the following options:
         1. Lufthansa
         2. Ryanair
user:    Lufthansa
```

The chatbot asks a question and specifies a list of possible options. The user responds by writing one among those listed names or by writing the ordinary number in front of the corresponding name. For the sake of brevity, we decided to present only two possible options (two airlines) to choose from. This statement is similar to the “switch” statement in programming languages. As the result of the execution of this part of the chatbot the output graph can be extended with the following triple:

```
foo:LH1234
   tio:operatedBy <http://dbpedia.org/resource/Lufthansa>.
```

The segment of the model that generates the previous chatbot question and adds the specified triple, as the consequence of the chosen option, is represented in Fig. 4. It can be observed that the previous conversational block *block_ 1* is used, as it is regarded a part of the same conversation section. The question to choose one out of several possible values is denoted by an instance of the obop:Question class, called *question_ 1*. *question_ 1* has the data property *obop:hasText* with the text of the question. Additionally, the question has the value of 2 for the *obop:hasPositionNumber* data property, which serves to denote that this question has second place in the conversation block. The question *question_ 1* has an instance of the class *obop:hasBranching* called *branching_ 1* and this branching is specified by the instance of the *obop:Connection* class named *conn_ 1*.

The obop:Connection class specifies how the two instances of the ontology classes should be related. An instance of this class has functional properties that specify the source and destination of object properties. In our case, the source is an instance called *flight_model_1* and the destination is *airline_model_1*. The latter serves as a model for an instance of the *dbpedia:Agent* class, which will be created during the interaction with the chatbot. Additionally, the object property that has to be inserted in the output graph between these two instances is specified by the *obop:containsDatatype* object property and it is, in our context, *tio:operatedBy* property. This particular object property is denoted by a dashed line in Fig. 4, although this edge is not part of the actual model graph.

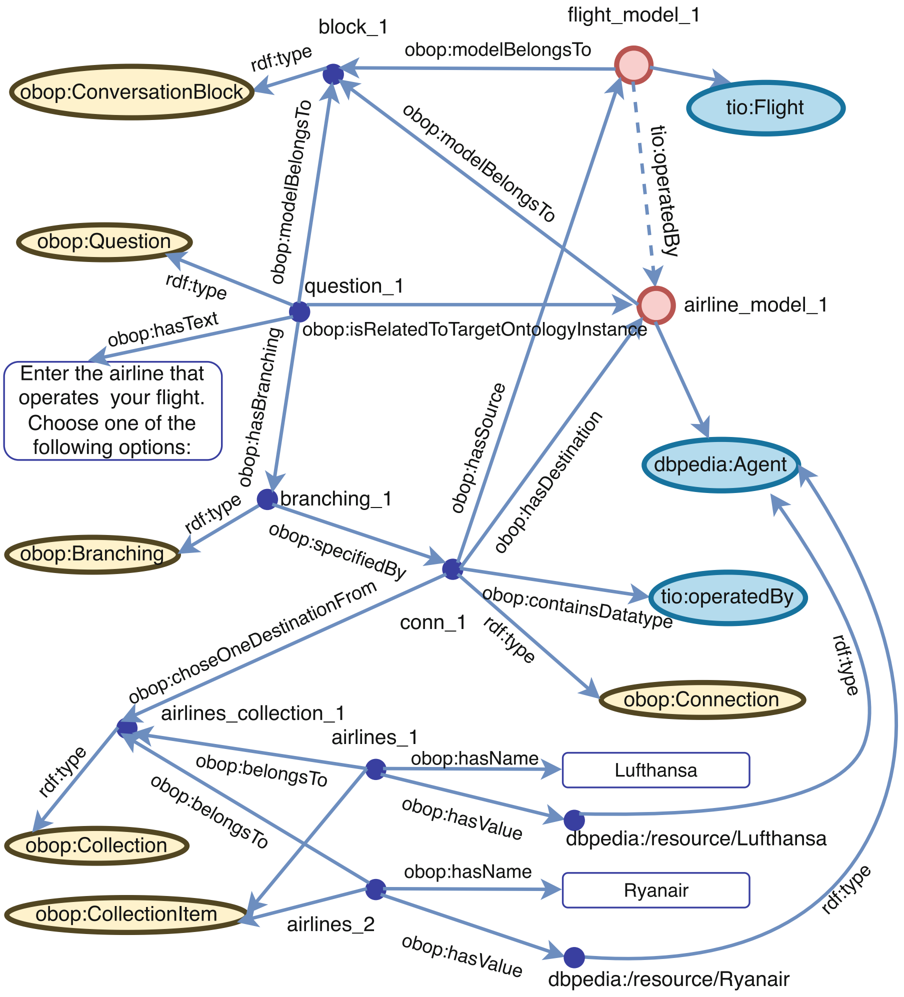

Fig. 4. A part of the model specifying the process of selecting an airline

It can be seen that the connection instance specifies both a predicate (object property) and an object (instance) of a new triple. The user, interacting with the chatbot, is required to choose a single airline from the list of possible airlines. In the lower section of Fig. 4 is defined an instance of the *obop:Collection* class, named *airlines_ collection_ 1*. The connection instance conn_1 is related to this collection by the object property *obop:choseOneDestinationFrom*, which specifies exactly how to form the object of a new triple. In this case, our simple collection contains two instances of the class *obop:CollectionItem* corresponding to Lufthansa and Ryanair airlines. OBOP collections are not defined using rdf:Bag, rdf:Seq or by using restrictions like *owl:someValuesFrom*. Collections are basically used by the chatbot program in the runtime phase and they are not meant to be used in the process of ontology reasoning.

An alternative, that can be specified in our model, which brings more generality to this example, is to allow the user to directly input an IRI of the desired airline, which must belong to the *dbpedia:Agent* class. However, we presented the selection from the list of available options to show that the model designer has this option.

4.3 Iteration Control Structure

To effectively manage control flows, the chatbot model has to provide definition of iteration structures (loops). As the iteration structures in programming languages allow the repetitive execution of a specific code block, iteration structures in chatbots enable the repetition of a set of questions within a conversation block or the repetition of other activities. For this purpose, the OBOP ontology introduces the *obop:Loop* class.

An example of iterations can be demonstrated within our flight scenario. The user has to define various ticket types for the flight, which could initiate the following conversation:

```
chatbot: Do you want to define one more ticket type?
user:    Yes
chatbot: Please enter the label of the ticket?
user:    Economy tickets from Frankfurt to London Heathrow
chatbot: Choose the service level of the ticket?
user:    Economy
chatbot: Do you want to define one more ticket type?
```

The result of executing the previous chatbot conversation is the following part of the graph:

```
foo:ticket5 a tio:TicketPlaceholder ;
  rdfs:label "Economy tickets from Frankfurt to London
              Heathrow"@en ;
  tio:scope [ a tio:ScopeOfAccess ;
     tio:accessTo foo:LH1234 ;
     tio:eligibleServiceLevel tio:Economy ] .
```

The part of the model that generates the preceding section of the chatbot conversation is illustrated in Fig. 5. The question *question_2*, which ask for a new ticket type is related to an instance of the *obop:Loop* class called *loop_1*. Each iteration of the loop specifies adding a new ticket which is represented by a conversational block *block_3*. If the user answers positively to the chatbot's question, a new ticket instance is generated according to the *ticket_model_1*. Together with the ticket is generated a blank node corresponding to the *_:bnode* instance. The new blank node is related to the flight instance, which is already created in the previous examples according to the *flight_model_1*.

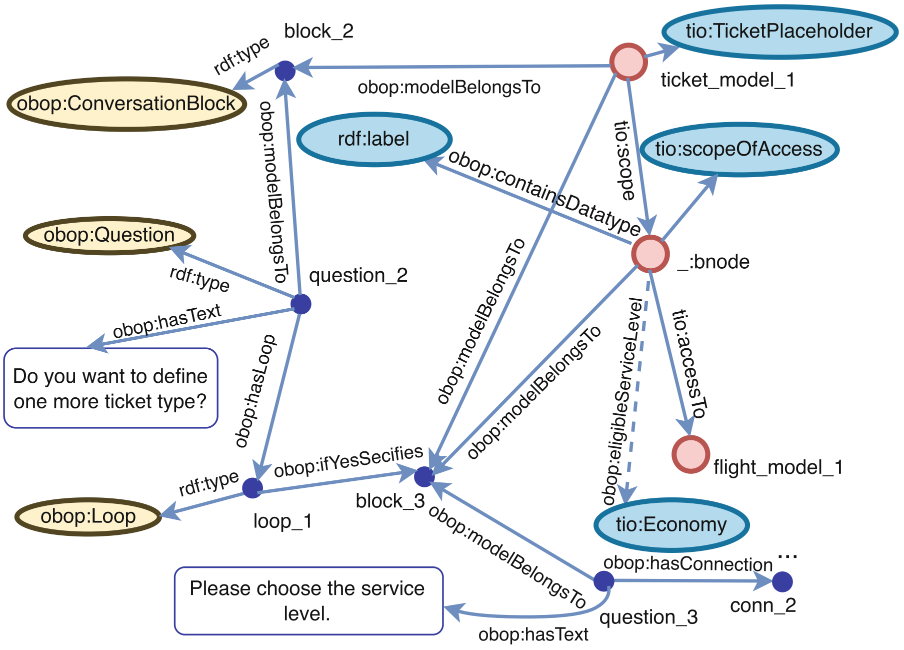

Fig. 5. A part of the model used to define tickets using iteration.

The object properties *tio:scope* and *tio:accessTo* are also automatically created. However, *rdfs:label* had to be specified using *obop:containsDatatype* object property. In order to choose between possible service classes (e.g., Business or Economy) a new question (*question_3*) and connection (*conn_2*) was necessary. *conn_2* is not completely presented and ontology classes of same instances are omitted in Fig. 5 for improved readability.

5 Implementation

To implement a prototype of our chatbot, we employed Python libraries, namely, *ChatterBot* as the conversational dialogue engine and *Owlready* [8] for ontology-oriented programming, to facilitate the creation and manipulation of OWL ontologies. The chatbot prototype can be tested as a REST application implemented using Flask framework[3] with a simple JavaScript frontend. The source code can be found on GitHub[4].

Chatbots generated using the ChatterBot library answer user questions based on the functionalities of logic adapters. For example, there are adapters like the *Time adapter* that can answer questions like "What time is it?", the *Math adapter* that can handle arithmetic operations, etc. When the user submits a question, all logic adapters assess whether they are capable of providing a response. The ability of a logical adapter to answer the question is measured by a confidence factor, expressed as a decimal value between 0 and 1. The answer from the adapter with the highest confidence is chosen as the reply. The architecture of our chatbot is shown in Fig. 6.

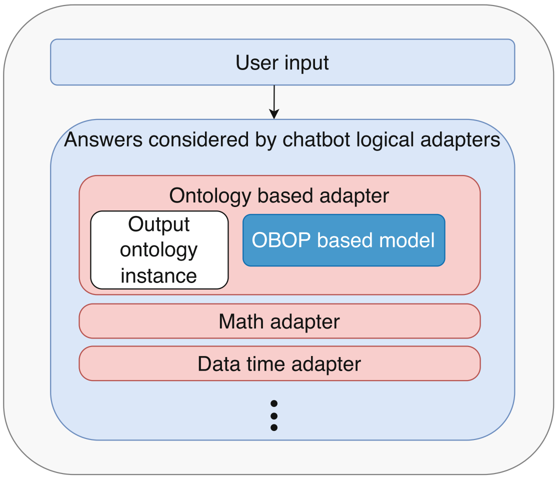

Fig. 6. Architecture of the chatbot system.

Our model-based chatbot functionalities are implemented as a logic adapter for the ChatterBot library, named the *OntoBasedAdapter*. Unlike other logic adapters in ChatterBot, the OntoBasedAdapter keeps track of the current state of conversation. Once the OntoBasedAdapter starts to gather information and populate the corresponding output knowledge graph, it needs to store information of the conversation's current state. It includes tracking visited nodes to enable system to make reverse steps and undo entries if necessary. Chatbot stores information on the current context, which is comprised of the entire array of nodes that have been visited and created during the conversation up to the current state.

[3] https://flask.palletsprojects.com.
[4] https://github.com/ontosoft/ontochatbot.

Keeping track of the state of an ongoing dialogue is quite challenging task. In the study [4], authors dealt with this problem by exploiting ontologies for both the knowledge base and the dialogue manager in domain-driven conversation, creating a banking chatbot. Our chatbot, on the other hand, is a domain independent conversational agent, with the domain specified in the corresponding chatbot model.

Answering Questions and Generating Replies. When the OntoBasedAdapter takes charge of the dialogue, it maintains the conversation status. The system asks questions in the order that is defined by the chatbot model. If the user asks a question that is not related to the current task specified in the model the conversation might take a different direction, potentially allowing another logic adapter to respond, enhancing the conversation's natural flow. Subsequently, it is posed a question by OntoBasedAdapter to pick up the conversation at the point where it was interrupted. This capability stems from the OntoBasedAdapter's management of context and conversation state. The confidence factor determines what logic adapter among all logical adapters will be chosen to give the answer. However, if the OntoBasedAdapter already started to gather information according to a conversation model then it has precedence over other logical adapters. Even if a user abruptly shifts the conversation's focus, the conversation agent tries to steer it back to the desired topic. For instance, if a user asks "What time is it?" during a conversation meant to schedule flight details, the system employs the *Time logic adapter* to answer promptly, before returning to the next question in line based on the chatbot model.

In this way, responses (questions) of any logic adapter are assigned a corresponding weight (confidence factor). The challenges associated to this way of choosing replies include the fact that confidence factors of already implemented adapters tend to be quite high values. The solution which we use is that OntoBasedAdapter always returns the confidence factor equal to 1 after it started to collect information. At present, OntoBasedAdapter responses are manually embedded into the model by the model designer.

6 Contribution

In this paper, we introduced models for conversational agents (chatbots). These chatbots can accomplish domain-specific tasks such as generating flight descriptions, booking flight tickets or creating restaurant menus that could be used for restaurant reservations in a user-friendly manner. In order to show how to create a chatbot model, we designed a chatbot intended for flight descriptions. The chatbots we propose follow the rule-based approach, in which replies are generated using predefined rules integrated into the models of respective conversations. These conversation models represent templates according to which replies (questions) are created.

The requirements for the approach outlined in the introduction of the paper are met with our implementation. The conversation model and the structure of the knowledge graph are both included in the same knowledge graph according to the first requirement. The limitations of this approach are the high complexity of the model graphs and challenging maintenance of the generated graphs. The second and third requirements are addressed by the explicit definition of specific classes, object properties, and data properties in the OBOP ontology. These definitions correspond to fundamental programming control structures, ensuring the capability to capture and manage various aspects of the conversation's flow and logic.

Describing chatbot models based on OWL ontologies might offer significant benefits, including the potential for OWL reasoning applied to these models. The OWL reasoning capability could identify flawed models that might result in inconsistent knowledge graphs. Another advantage lies in the potential for reusing existing models, enabling their sharing and querying through SPARQL. Moreover, the application of machine learning algorithms to existing models could streamline the process of automatically or semi-automatically generating new chatbots for description of similar problems.

References

1. Agarwal, R., Wadhwa, M.: Review of state-of-the-art design techniques for chatbots. SN Comput. Sci. **1**(5), 246 (2020)
2. Ait-Mlouk, A., Jiang, L.: KBot: a knowledge graph based chatbot for natural language understanding over linked data. IEEE Access **8**, 149220–149230 (2020)
3. Al-Zubaide, H., Issa, A.A.: OntBot: ontology based chatbot. In: International Symposium on Innovations in Information and Communications Technology, pp. 7–12. IEEE (2011)
4. Altinok, D.: An ontology-based dialogue management system for banking and finance dialogue systems. arXiv preprint arXiv:1804.04838 (2018)
5. Bocklisch, T., Faulkner, J., Pawlowski, N., Nichol, A.: Rasa: open source language understanding and dialogue management. arXiv preprint arXiv:1712.05181 (2017)
6. Hepp, M.: GoodRelations: an ontology for describing products and services offers on the web. In: Gangemi, A., Euzenat, J. (eds.) EKAW 2008. LNCS (LNAI), vol. 5268, pp. 329–346. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87696-0_29
7. Hitz, M., Radonjic-Simic, M., Reichwald, J., Pfisterer, D.: Generic UIs for requesting complex products within distributed market spaces in the internet of everything. In: Buccafurri, F., Holzinger, A., Kieseberg, P., Tjoa, A.M., Weippl, E. (eds.) CD-ARES 2016. LNCS, vol. 9817, pp. 29–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45507-5_3
8. Lamy, J.B.: Owlready: ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. Artif. Intell. Med. **80**, 11–28 (2017)
9. Pérez-Soler, S., Guerra, E., de Lara, J.: Model-driven chatbot development. In: Dobbie, G., Frank, U., Kappel, G., Liddle, S.W., Mayr, H.C. (eds.) ER 2020. LNCS, vol. 12400, pp. 207–222. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62522-1_15
10. Radonjic-Simic, M., Pfisterer, D., Rutesic, P.: Arising internet of everything: Business modeling and architecture for smart cities in recent developments in engineering research, vol. 8, chap. 7 (2020)

11. Rutesic, P., Radonjic-Simic, M., Pfisterer, D.: An enhanced meta-model to generate web forms for ontology population. In: Villazón-Terrazas, B., Ortiz-Rodríguez, F., Tiwari, S., Goyal, A., Jabbar, M.A. (eds.) KGSWC 2021. CCIS, vol. 1459, pp. 109–124. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91305-2_9
12. Turnip, T.N., Silalahi, E.K., Sinulingga, Y.A.V., Siregar, V.: Application of ontology in semantic web searching of flight ticket as a study case. J. Phys. Conf. Ser. **1175**, 012092 (2019)
13. Wallace, R.S.: The anatomy of A.L.I.C.E. In: Epstein, R., Roberts, G., Beber, G. (eds.) Parsing the Turing Test, pp. 181–210. Springer, Dordrecht (2009). https://doi.org/10.1007/978-1-4020-6710-5_13