Towards Automating the Synthesis of Chatbots for Conversational Model Query

Sara Pérez-Soler[1], Gwendal Daniel[2], Jordi Cabot[2,3], Esther Guerra[1(✉)], and Juan de Lara[1]

[1] Universidad Autónoma de Madrid, Madrid, Spain
{Sara.PerezS,Esther.Guerra,Juan.deLara}@uam.es
[2] Universitat Oberta de Catalunya, Barcelona, Spain
gdaniel@uoc.edu
[3] ICREA, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. Conversational interfaces (also called chatbots) are being increasingly adopted in various domains such as e-commerce or customer service, as a direct communication channel between companies and end-users. Their advantage is that they can be embedded within social networks, and provide a natural language (NL) interface that enables their use by non-technical users. While there are many emerging platforms for building chatbots, their construction remains a highly technical, challenging task.

In this paper, we propose the use of chatbots to facilitate querying domain-specific models. This way, instead of relying on technical query languages (e.g., OCL), models are queried using NL as this can be more suitable for non-technical users. To avoid manual programming, our solution is based on the automatic synthesis of the model query chatbots from a domain meta-model. These chatbots communicate with an EMF-based modelling backend using the Xatkit framework.

Keywords: Model-driven engineering · Model query · Automatic chatbot synthesis

1 Introduction

Instant messaging platforms have been widely adopted as one of the main technologies to communicate and exchange information. Most of them provide built-in support for integrating *chatbot applications*, which are automated conversational agents capable of interacting with users of the platform [10]. Chatbots have proven useful in various contexts to automate tasks and improve the user experience, such as automated customer services [23], education [9] and e-commerce [21]. However, despite many platforms have recently emerged for creating chatbots (e.g., DialogFlow [6], IBM Watson [7], Amazon Lex [1]), their construction and deployment remains a highly technical task.

S. Nurcan et al. (Eds.): BPMDS 2020/EMMSAD 2020, LNBIP 387, pp. 257–265, 2020.
https://doi.org/10.1007/978-3-030-49418-6_17

Chatbots are also increasingly used to facilitate software engineering activities [5,12] like automating deployment tasks, assigning software bugs and issues, repairing build failures, scheduling tasks like sending reminders, integrating communication channels, or for customer support. In this context, we explored the use of chatbots for domain modelling in previous work [16,17]. Modelling chatbots can be embedded within social networks to support collaboration between different stakeholders in a natural way, and enable the active participation of non-technical stakeholders in model creation.

In the present work, we extend the previous ideas to support natural language (NL) conversational queries over the models. This is a more accessible and user-friendly way to query models than the use of technical languages like OCL (Object Constraint Language [15]). Moreover, we avoid the manual programming of the model query chatbots by their automatic synthesis. For this purpose, our solution is based on (i) the availability of a meta-model describing the structure of the models, (ii) its configuration with NL information (class name synonyms, names for reverse associations, etc.), and (iii) the automatic generation of a chatbot supporting queries over instances of the given meta-model. This approach is implemented on top of the Xatkit model-based chatbot development platform [4], which interprets the generated chatbot model and interacts with an EMF (Eclipse Modeling Framework) backend.

The rest of the paper is structured as follows. First, Sect. 2 provides motivation using a running example, and introduces background about chatbot design. Then, Sect. 3 explains our approach, and Sect. 4 describes the prototype tool support. Finally, Sect. 5 compares with related works, and Sect. 6 concludes.

2 Motivation and Background

In this section, we first provide a motivating example, and then introduce the main concepts behind chatbots.

2.1 Motivation

As a motivating example, assume a city hall would like to provide open access to its real-time traffic information system. Given the growth of the open data movement, this is a common scenario in many cities, like Barcelona[1] or Madrid[2].

We assume that the data provided includes a static part made of the different districts and their streets, with information on the speed limits. In addition, a dynamic part updated in real-time decorates the streets and their segments with traffic intensity values and incidents (road works, street closings, accidents or bottlenecks). Figure 1 shows a meta-model capturing the structure of the provided information.

In this scenario, citizens would benefit from user-friendly ways to query those traffic models. However, instead of relying on the construction of dedicated front-ends with fixed queries, or on the use of complex model query languages like

[1] https://opendata-ajuntament.barcelona.cat/.
[2] https://datos.madrid.es.

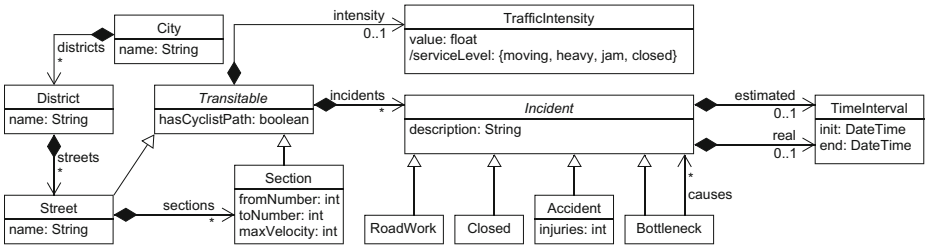

Fig. 1. A meta-model for real-time traffic information.

OCL, our proposal is the use of conversational queries based on NL via chatbots. Chatbots can be used from widely used social networks, like Telegram or Twitter, facilitating their use by citizens. Hence, citizens would be able to issue simple queries like *"give me all accidents with more than one injury"*; and also conversational queries like *"what are the incidents in Castellana Street now?"*, and upon the chatbot reply, focus on a subset of the results with *"select those that are accidents"*. Finally, for the case of dynamic models, reactive queries like *"ping me when Castellana Street closes"* would be possible.

Our proposal consists in the generation of a dedicated query chatbot given the domain meta-model. But, before introducing our approach, the next subsection explains the main concepts involved in chatbot design.

2.2 Designing a Chatbot

The widespread interest and demand for chatbot applications has emphasized the need to quickly build complex chatbots supporting NL processing (NLP) [8], custom knowledge base definition [18], and complex action responses including external service composition. However, the development of chatbots is challenging as it requires expertise in several technical domains, ranging from NLP to a deep understanding of the API of the targeted instant messaging platforms and third-party services to be integrated. To alleviate this situation, many chatbot creation frameworks have emerged, like DialogFlow [6], IBM Watson [7] or Amazon Lex [1].

Figure 2 shows a simplification of the typical working scheme of chatbots. Chatbots are often designed on the basis of *intents*, where each intent represents some user's aim (e.g., booking a ticket). The chatbot waits for NL inputs from the user (label 1 in the figure); then, it tries to match the phrase with some intent (label 2), optionally calling an external service (label 3) for intent recognition or additional data collection; finally, it produces a response, which is often a NL sentence among a predefined set (label 4).

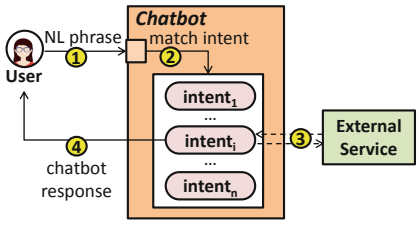

Fig. 2. Chatbot working scheme.

Intents are defined via training phrases. These phrases may include parameters of a certain type (e.g., numbers, days of the week, countries). The parameter types are called *entities*. Most platforms come with predefined sets of entities and permit defining new ones. Some platforms permit structuring the conversation as an expected flow of intents. For this purpose, a common mechanism is providing intents with a *context* that stores information gathered from phrase parameters, and whose values are required to trigger the intent. In addition, there is normally the possibility to have a *fallback* intent, to be used when the bot does not understand the user input.

3 Approach

Figure 3 shows the scheme of our approach. First, the chatbot designer needs to provide a domain meta-model (like the one in Fig. 1) defining the structure of the models to be queried, and complemented with NL hints on how to refer to its classes and features (synonyms). From this information, an executable chatbot model that can be used to query model instances is generated. The next subsections explain these two steps.

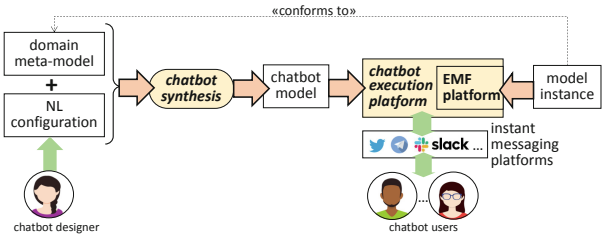

Fig. 3. Scheme of our approach.

3.1 Chatbot Generation: Intents and Entities Model

The chatbot designer has to provide a domain meta-model and optionally, a NL configuration model. The latter is used to optionally annotate classes, attributes and features with synonyms, and the source of references with a name to refer to its backward navigation. From this information, we generate the chatbot intents and entities.

Table(a) of Fig. 4 captures the generation of intents. We create an intent per query type, plus an additional intent called loadModel to select the model to be queried. The second row of the table shows the intent allInstances, which returns all objects of a given class. The intent is populated with training phrases that contain the class name as parameter. The possible class names are defined via an entity Class (see Table(b)). This intent would be selected on user utterances such as *"give me all cities"* or *"show every incident"*. The intent requires having a loaded model, which the table indicates as the intent requiring a model as context.

In the same table, intent filteredAllInstances returns all instances that satisfy a given condition. The intent is populated with training phrases that combine a class name and a condition made of one or more filters joined via logical

a) Intents

name	description	training phrases	provided context	required context
loadModel	loads working model from the backend	load the model {MODEL} open model {MODEL}...	MODEL type text	-
allInstances	returns all instances of a given class	give me all the {CLASSNAME} show me the {CLASSNAME}...	CLASSNAME type Class	MODEL
filtered AllInstances	returns all instances of a given class and satisfying a condition	select the {CLASSNAME} with {FILTER1} display the {CLASSNAME} with {FILTER1} {CONJ} {FILTER2}...	CLASSNAME type Class FILTER1 and FILTER2 type Condition CONJ type Conjunction	MODEL

b) Class entity

entries	synonyms
city	metropolis, town
...	...
bottleneck	traffic jam, congestion

c) StringAttribute entity

entries	synonyms
name	title, designation
description	summary

d) NumericAttribute entity

entries	synonyms
from number	from, starts
to number	to, ends
max velocity	velocity limit
value	amount of traffic
injuries	harm

e) NumericOperator entity

entries	synonyms
greater than	bigger, more than
smaller than	less than
equals	is same as

f) Condition composite entity

type	entries
StringCondition	StringAttribute + StringOperator + text
	StringAttribute + StringOperator + StringAttribute
NumericCondition	NumericAttribute + NumericOperator + number
	NumericAttribute + NumericOperator + NumericAttribute

g) Conjunction entity

entries
and
or

h) StringOperator entity

entries	synonyms
starts with	begins with
ends with	finishes with, end is
equals	is same as
contains	has

Fig. 4. Intents and entities generated for the running example chatbot.

connectives. We provide an entity Condition for the filters, explained below. This intent would be selected upon receiving phrases like *"give me all accidents with more than one injury"* (please note the singular variation w.r.t. the attribute name injuries).

In addition to intents, we create several entities based on the domain meta-model and the NL configuration. Specifically, we create an entity named Class (Table(b)) with an entry for each meta-model class name. These entries may have synonyms, as provided by the NL configuration, to refer to the classes in a more flexible way. Likewise, we create an entity for each attribute name attending to their type: String (Table(c)), Numeric (Table(d)), Boolean and Date (omitted for space constraints). For example, the StringAttribute entity (Table(c)) has an entry for all String attributes called name. Just like classes, these entries may have synonyms if provided in the NL configuration.

The Condition entity (Table(f)) is a composite one, i.e., its entries are made of one or more entities. This entity permits defining filter conditions in queries, such as *"name starts with Ma"* or *"injuries greater than one"*.

Regarding the complexity of the chatbot, the number of intents is fixed, and it depends on the primitives of the underlying query language that the chatbot exposes. Figure 4 exposes two primitives of OCL: allInstances, and allInstances()→select(cond). Other query types can be added similarly, which would require defining further intents. The number of generated entities is also fixed, while the number of entries in each entity depends on the meta-model size and the synonyms defined in the NL configuration.

3.2 Chatbot Generation: Execution Model

The generated chatbot also contains actions, required to perform the query on a modelling backend, which we call the *execution model*. This execution model

contains a set of *execution rules* that bind user intentions to response actions as part of the chatbot behaviour definition (cf. label 4 in Fig. 2). For each intent in the *Intent* model, we generate the corresponding execution rule in the execution model using an event-based language that receives as input the recognized intent together with the set of parameter values matched by the NL engine during the analysis and classification of the user utterance.

All the execution rules follow the same process: the matched intent and the parameters are used to build an OCL-like query to collect the set of objects the user wants to retrieve. The intent determines the type of query to perform (e.g., allInstances, select, etc.), while the parameters identify the query parameters, predicates, and their composition. The query computation is delegated to the underlying modelling platform (see next section), and the returned model elements are processed to build a human-readable message that is finally posted to the user by the bot engine.

As an example, Listing 1 shows the execution rule that handles an allInstances operation. The class to obtain the instances of is retrieved from the context variable (available in every execution rule) and passed to our EMF Platform, which performs the query. Next, the instances variable holding the results is processed to produce a readable string (in this case a list of names), and the Chat Platform is called to reply to the user.

```
1 on intent GetAllInstances do
2   val Map<String, Object> collectionContext = context.get("collection")
3   val instances = EMFPlatform.GetAllInstances( collectionContext.get("class") as String )
4   val resultString = instances.map[name].join(", ")
5   ChatPlatform.Reply("I found the following results" + resultString)
```

Listing 1. Execution rule example

4 Proof of Concept

As a proof of concept, we have created a prototype that produces Xatkit-based chatbots [4], following the two phases depicted in Fig. 3. Xatkit is a model-driven solution to define and execute chatbots, which offers DSLs to define the bot intents, entities and actions. The execution of such chatbots relies on the Xatkit *runtime* engine. At its core, the engine is a Java library that implements all the execution logic available in the chatbot DSLs. Besides, a connector with Google's DialogFlow engine [6] takes care of matching the user utterances, and a number of platform components enable the communication between Xatkit and other external services.

In the context of this paper, we have developed a new EMF Platform that allows Xatkit to query EMF models in response to matched intents. The first version of our prototype platform[3] provides actions to retrieve all the instances of a given class, and filter them based on a composition of boolean predicates on the object's attributes or references. These predicates are retrieved from the context parameter defined in the intents (see Sect. 3.1), and mapped

[3] https://github.com/xatkit-bot-platform/xatkit-emf-platform.

to Java operations (e.g., the StringComparison "contains" is translated into ((String)value).contains(otherValue). The query result is returned as a list of EObjects, which is processed using the bot expression language to produce the response message. Listing 1 showed an example of the use of this EMF Platform.

We have also developed a web application, where domain meta-models (in .ecore format) can be uploaded, and then (optionally) configured with synonyms. Once the configuration is finished, the application synthesizes a Xatkit chatbot model, which then can be executed using the Xatkit runtime engine.

Figure 5(a) shows the web application on the left, where the running example meta-model (cf. Fig. 2) is being configured. Figure 5(b) shows a moment in the execution of the generated Xatkit chatbot, and the result returned by the bot when processing the example utterance *"show all accidents with more than one injury"*.

Fig. 5. (a) Web application to configure the chatbot. (b) A query in the generated chatbot.

5 Related Work

Next, we review approaches to the synthesis of chatbots for modelling or data query.

Our work relies on NL as a kind of concrete syntax for DSLs [17]. NLP has been used within Software Engineering to derive UML diagrams/domain models from text [2,11]. However, the opposite direction (i.e., generating chatbots from domain models) is largely unexplored. Almost no chatbot platform supports automatic chatbot generation from external data sources. A relevant exception is Microsoft QnA Maker [14], which generates bots for the Azure platform from FAQs and other well-structured textual information.

Closest approaches to ours are tools like ModelByVoice [13] and VoiceToModel [20], which offer some predefined commands to create model elements for specific types of models. In contrast, our framework targets model queries and not model creation, which was pursued in our previous work [17]. None of those two approaches support queries. Castaldo and collaborators [3] propose generating chatbots for data exploration in relational databases, but requiring an annotated schema as starting point, while in our case providing synonyms is an optional step. Similarly, [19] integrates chatbots to service systems by annotating and linking the chatbot definition to the service models. In both cases, annotations and links must be manually created by the chatbot designer to generate the conversational elements. In contrast, our approach is fully automatic. In [22], chatbots are generated from OpenAPI specifications but the goal of such chatbots is helping the user in identifying the right API Endpoint, not answering user queries.

Altogether, to our knowledge there are no automatic approaches to the generation of flexible chatbots with model query capabilities. We believe that applying classical concepts from CRUD-like generators to the chatbot domain is a highly novel solution to add a conversational interface to any modelling language.

6 Conclusion

Conversational interfaces are becoming increasingly popular to access all kind of services, but their construction is challenging. To remedy this situation, we have proposed the automatic synthesis of chatbots able to query the instances of a domain meta-model.

In the future, we aim to support more complex queries, including the conversational and reactive ones mentioned in Sect. 2.1. Our approach could be used to query other types of data sources (e.g., databases or APIs) via an initial reverse engineering step to build their internal data model and translate the NL query into the query language of the platform. Finally, we would like to add access control on top of the bot definition to ensure users cannot explore parts of the model/system unless they have permission.

Acknowledgments. Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00 and TIN2016-75944-R) and the R&D programme of Madrid (P2018/TCS-4314).

References

1. Amazon: Amazon Lex (2019). https://aws.amazon.com/lex/
2. Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F.: Extracting domain models from natural-language requirements: approach and industrial evaluation. In: Proceedings of the MoDELS, pp. 250–260. ACM (2016)
3. Castaldo, N., Daniel, F., Matera, M., Zaccaria, V.: Conversational data exploration. In: Bakaev, M., Frasincar, F., Ko, I.-Y. (eds.) ICWE 2019. LNCS, vol. 11496, pp. 490–497. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19274-7_34

4. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: a multimodal low-code chatbot development framework. IEEE Access **8**, 15332–15346 (2020)
5. Erlenhov, L., de Oliveira Neto, F.G., Scandariato, R., Leitner, P.: Current and future bots in software development. In: Proceedings of the BotSE@ICSE, pp. 7–11. IEEE/ACM (2019)
6. Google: DialogFlow (2019). https://dialogflow.com/
7. IBM Watson Assistant (2019). https://www.ibm.com/cloud/watson-assistant/
8. Jackson, P., Moulinier, I.: Natural Language Processing for Online Applications: Text Retrieval, Extraction and Categorization, vol. 5. John Benjamins Publishing, Amsterdam (2007)
9. Kerlyl, A., Hall, P., Bull, S.: Bringing chatbots into education: towards natural language negotiation of open learner models. In: Ellis, R., Allen, T., Tuson, A. (eds.) SGAI 2006, pp. 179–192. Springer, London (2007). https://doi.org/10.1007/978-1-84628-666-7_14
10. Klopfenstein, L., Delpriori, S., Malatini, S., Bogliolo, A.: The rise of bots: a survey of conversational interfaces, patterns, and paradigms. In: Proceedings of the DIS, pp. 555–565. ACM (2017)
11. Landhäußer, M., Körner, S.J., Tichy, W.F.: From requirements to UML models and back: how automatic processing of text can support requirements engineering. Softw. Qual. J. **22**(1), 121–149 (2014)
12. Lebeuf, C., Storey, M.D., Zagalsky, A.: Software bots. IEEE Softw. **35**(1), 18–23 (2018)
13. Lopes, J., Cambeiro, J., Amaral, V.: ModelByVoice - towards a general purpose model editor for blind people. In: Proceedings of the MODELS Workshops. CEUR Workshop Proceedings, vol. 2245, pp. 762–769. CEUR-WS.org (2018)
14. Microsoft: QnA Maker (2019). https://www.qnamaker.ai/
15. OCL (2014). http://www.omg.org/spec/OCL/
16. 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)
17. Pérez-Soler, S., González-Jiménez, M., Guerra, E., de Lara, J.: Towards conversational syntax for domain-specific languages using chatbots. JOT **18**(2), 5:1–21 (2019)
18. Shawar, A., Atwell, E., Roberts, A.: FAQchat as in information retrieval system. In: Proceedings of the LTC, pp. 274–278. Wydawnictwo Poznańskie, Poznań (2005)
19. Sindhgatta, R., Barros, A., Nili, A.: Modeling conversational agents for service systems. In: Panetto, H., Debruyne, C., Hepp, M., Lewis, D., Ardagna, C.A., Meersman, R. (eds.) OTM 2019. LNCS, vol. 11877, pp. 552–560. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33246-4_34
20. Soares, F., Araújo, J., Wanderley, F.: VoiceToModel: an approach to generate requirements models from speech recognition mechanisms. In: Proceedings of the SAC, pp. 1350–1357. ACM (2015)
21. Thomas, N.: An e-business chatbot using AIML and LSA. In: Proceedings of the ICACCI, pp. 2740–2742. IEEE (2016)
22. Vaziri, M., Mandel, L., Shinnar, A., Siméon, J., Hirzel, M.: Generating chat bots from web API specifications. In: Proceedings of the ACM SIGPLAN Onward!, pp. 44–57 (2017)
23. Xu, A., Liu, Z., Guo, Y., Sinha, V., Akkiraju, R.: A new chatbot for customer service on social media. In: Proceedings of the CHI, pp. 3506–3510. ACM (2017)