Model-Driven Chatbot Development

Chapter · October 2020

DOI: 10.1007/978-3-030-62522-1_15

3 authors:

Sara Pérez-Soler
Universidad Autónoma de Madrid
16 PUBLICATIONS **234** CITATIONS
SEE PROFILE

Esther Guerra
Universidad Autónoma de Madrid
195 PUBLICATIONS **3,364** CITATIONS
SEE PROFILE

Juan de Lara
Universidad Autónoma de Madrid
370 PUBLICATIONS **6,138** CITATIONS
SEE PROFILE

Model-driven chatbot development

Sara Pérez-Soler[0000−0002−4558−7111], Esther Guerra[0000−0002−2818−2278], and Juan de Lara[0000−0001−9425−6362]

Universidad Autónoma de Madrid (Spain)
(Sara.PerezS, Esther.Guerra, Juan.deLara)@uam.es

Abstract. Chatbots are software services accessed via conversation in natural language. They are increasingly used to help in all kinds of procedures like booking flights, querying visa information or assigning tasks to developers. They can be embedded in webs and social networks, and be used from mobile devices without installing dedicated apps. While many frameworks and platforms have emerged for their development, identifying the most appropriate one for building a particular chatbot requires a high investment of time. Moreover, some of them are closed – resulting in customer lock-in – or require deep technical knowledge.
To tackle these issues, we propose a model-driven engineering approach to chatbot development. It comprises a neutral meta-model and a domain-specific language (DSL) for chatbot description; code generators and parsers for several chatbot platforms; and a platform recommender. Our approach supports forward and reverse engineering, and model-based analysis. We demonstrate its feasibility presenting a prototype tool and an evaluation based on migrating third party Dialogflow bots to Rasa.

Keywords: Chatbots · Model-Driven Engineering · DSLs · Migration

1 Introduction

Chatbots are software programs that interact with users via natural language (NL) conversation. Their use is booming because they can be used within webs and social networks – like Telegram, Twitter or Slack – without having to install dedicated apps [23]. Many companies are developing chatbots to offer 24/7 customer service while reducing costs, and their presence is percolating a wide range of areas such as education [26, 29, 30] or civic engagement [27].

The success of chatbots has led to the emergence of a plethora of technologies for their creation. Not only big software companies have made available chatbot creation tools, like Google's Dialogflow [9], IBM's Watson Assistant [28], Microsoft's bot framework [17] or Amazon's Lex [15], but many other proposals exist, like Rasa [21], FlowXO [10] and Pandorabots [18]. Among them, we find a variety of approaches. For example, Dialogflow and Watson offer low-code cloud development platforms that support the creation and deployment of bots, while Rasa is a framework that requires Python programming for bot development.

Overall, these chatbot creation tools are indisputably powerful (e.g., some provide NL processing, speech recognition, etc.). However, since there are so

many options, choosing the most appropriate one to develop a chatbot with certain features is not easy. There may also be operational factors to consider in the decision, as for example, some options may imply vendor lock-in, and migrating chatbots between tools is not generally supported. Last but not least, some approaches have a steep learning curve and require expert knowledge.

To overcome these problems, we propose a model-driven engineering (MDE) approach [22] to chatbot development. This relies on a meta-model with core primitives for chatbot design, and a domain-specific language (DSL) to define bots independently of the implementation technology. Chatbots defined with the DSL can be analysed for "smells" of defects, and a ranked list of appropriate bot creation tools is recommended based on the chatbot definition and other requirements. Our DSL can be used for *forward engineering*, to produce the chatbot implementation from its specification; and for *reverse engineering*, to produce a model out of a chatbot implementation, which can then be analysed, refactored and migrated to other platforms. Currently, we provide code generators and parsers from/to Dialogflow and Rasa, but our architecture is extensible. We evaluate our approach migrating third-party Dialogflow chatbots to Rasa.

In the rest of the paper, Section 2 introduces chatbot design and motivates our work. Section 3 outlines our proposal. Section 4 describes the meta-model and the DSL. Section 5 details our platform recommender. Section 6 presents tool support. Section 7 reports an evaluation based on migration. Section 8 compares with related works, and Section 9 concludes and outlines future work.

2 Building a chatbot: background and limitations

Chatbots (also called conversational agents) are software programs with a conversational user interface. They can be classified into *open-domain*, if they can converse on any topic with users, or *task-specific*, if they assist in a concrete task (e.g., bookings flights or shopping). Our work targets the latter kind of bots.

Fig. 1 shows the typical working scheme of task-specific chatbots. They are designed around a set of *intents* that users may want to accomplish. Given a user utterance (e.g., *"I'd like to buy a flight ticket from Madrid to Vienna"*, label 1 in the figure), the chatbot tries to identify the corresponding intent (label 2). The approach for this depends on the particular chatbot creation tool. Some of them – like Pandorabots – permit defining patterns or regular expressions upon which the utterance is matched, while others – like Dialogflow, Lex or Rasa – require declaring training phrases and apply NL processing (NLP) techniques. If the chatbot does not find any matching intent, some approaches

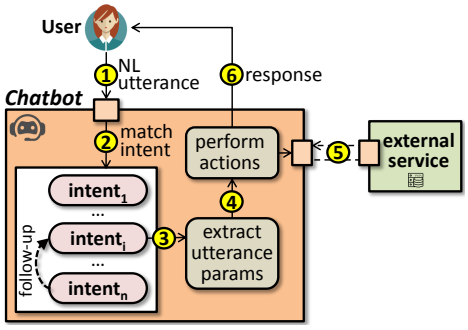

Fig. 1: Chatbot working scheme.

allow having a default fallback intent. In addition, the conversation flow can be structured into expected sequences of intents (relation *follow-up* in the figure).

After matching an intent, the chatbot extracts the *parameters* of interest from the utterance (e.g., the origin and destination of the flight, label 3). Parameters may be typed by *entities*, which can be either predefined (e.g., date, number) or specific to a chatbot (e.g., flight class). If the utterance lacks some expected parameters (e.g., date of flight), the chatbot can be configured to ask for them.

As a last step, the chatbot can perform different actions depending on the intent, such as calling an external service (e.g., a booking information system, label 5) or replying to the user (label 6). The simplest response format is text, but some chatbot deployment platforms (e.g., Telegram, Twitter) also support images, URLs, videos or buttons.

There are numerous tools for creating chatbots that follow this scheme. These tools use different approaches, ranging from low-code form-based platforms (e.g., Dialogflow, Lex, Watson, FlowXO) to frameworks for programming languages (e.g., Rasa, Botkit [4]), libraries (e.g., Chatterbot [6]) and services (e.g., LUIS [16]). Such a variety makes it difficult to ascertain which tool is suitable to build a specific chatbot, as not every tool supports every possible feature (e.g., only a few provide NLP or multi-language support). Moreover, the conceptual model of the chatbot might be difficult to attain, as the chatbot definition frequently includes tool-specific accidental details. As a consequence, reasoning, understanding, validating and testing chatbots independently from the implementation technology becomes challenging. Finally, some platforms are proprietary which hinders chatbot migration and results in vendor lock-in.

In the following section, we present our proposal to overcome these problems.

3 Model-driven engineering of chatbots

Fig. 2 shows a scheme of our proposal. It provides a technology-agnostic DSL called Conga (ChatbOt modelliNg lanGuAge) to design chatbots. This is built on the basis of a neutral, platform-independent meta-model resulting from an analysis of the existing approaches. The DSL permits modelling chatbots independently of any development platform, and validating quality criteria and well-formedness rules on the chatbot models. Section 4 introduces this DSL.

To facilitate the task of selecting a development tool for implementing a given chatbot model, we provide an extensible recommender that analyses the chatbot model as well as other requirements, to provide a ranked list of suitable tools. Section 5 explains the recommender system and its extensible architecture.

In addition, the DSL is complemented with code generators that synthesize chatbot implementations from chatbot models for specific development tools (e.g., JSON configuration files in the case of Dialogflow, or Python programs and configuration files in the case of Rasa). The chatbots so generated can be deployed in different platforms (e.g., Telegram, Slack or Twitter) to make them available to users. Likewise, the DSL facilitates chatbot migration by the provision of parsers from several development platforms into the DSL. Our tool

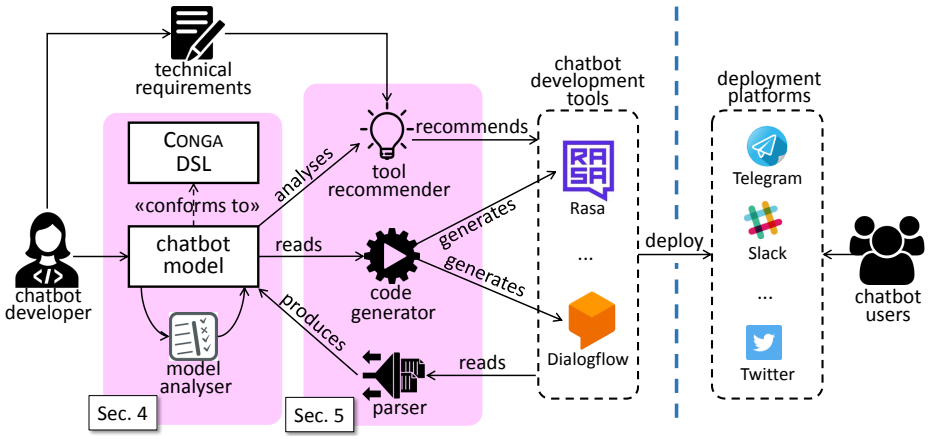

Fig. 2: Overview of our proposal.

support for these scenarios is explained in Section 6, while its evaluation based on migration scenarios is presented in Section 7.

Overall, the advantages of our proposal are the following: it keeps the design of the chatbot independent of the specific development technology; it provides analyses applicable at the design level (i.e., prior to the implementation); it assists in the selection of an appropriate development tool; it enables both forward and backward engineering; and it reduces the risk of vendor lock-in.

4 CONGA: a DSL for chatbot design

Our DSL CONGA enables the design of chatbots conformant to the neutral meta-model of Fig. 3. This is a platform-independent meta-model which gathers recurrent concepts in chatbot development approaches. Table 1 summarizes the main concepts of the 15 approaches that we have revised to design our meta-model.

The main meta-model class is *Chatbot*, which has a *name* and a list of supported *languages* to allow the definition of multi-language chatbots. Chatbots can define *intents*, *entities*, *actions* and structure the dialogue via *flows*.

Most analysed approaches (10 out of 15) rely on the notion of intent. In our meta-model, an *Intent* has a *name*, can be a *fallback* intent, and defines one set of regular expressions or NL training phrases per supported language. As Table 1 shows (3^{rd} and 4^{th} columns), all approaches support at least one of these two definition mechanisms, while 6 approaches can combine regular expressions with NL phrases. An example of a training phrase in English to query the price of a cake can be *"How much does a chocolate cake cost?"*.

Intents may need to collect information, like the cake flavour in the previous sentence. This information is stored in *Parameter*s, which most approaches support (see 5^{th} column of Table 1). In our meta-model, *Parameter*s have a *name*, a *type*, can be a *list*, can be *required*, and may define a list of *prompts* to

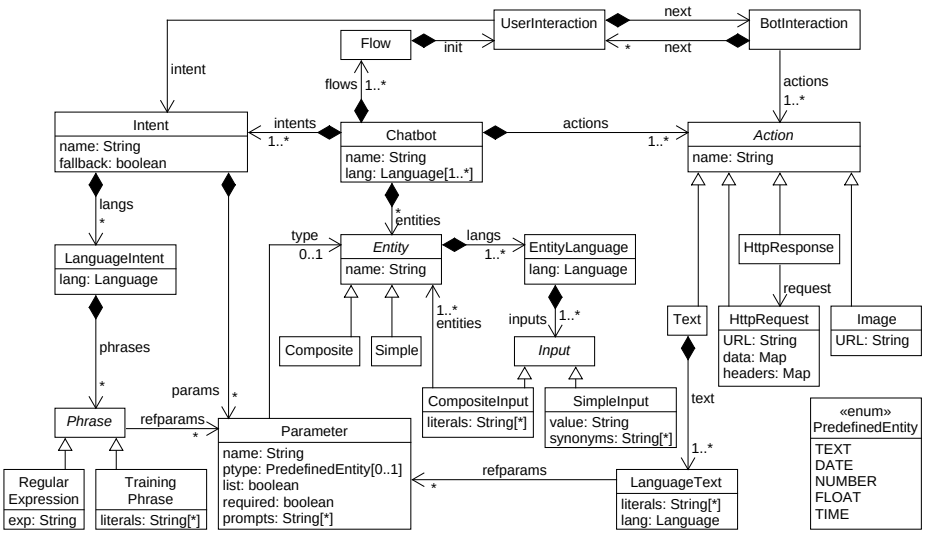

Fig. 3: Platform-independent chatbot design meta-model (simplified excerpt).

Table 1: Recurrent concepts of representative chatbot creation approaches.

Approach	**Intent**	**NLP**	**Regular expr**	**Phrase params**	**Entities**	**Answ. Text**	**Answ. Image**	**Http Rq/Rs**	**Dialogue structure**
Botkit [4]	no	no	yes	no	no	no	no	no	programm.
Bot framework [17]	yes	yes	yes	yes	yes	yes	yes	yes	tree
Chatfuel [5]	no	yes	no	no	no	yes	yes	yes	tree
Chatterbot [6]	no	yes	no	no	yes	no	no	no	context
Dialogflow [9]	yes	yes	no	yes	yes	yes	yes	yes	context
FlowXO [10]	yes	no	yes	no	yes	yes	yes	yes	tree
Landbot.io [14]	no	no	yes	no	no	yes	yes	yes	tree
Lex [15]	yes	yes	no	yes	yes	yes	yes	yes	session
LUIS [16]	yes	yes	yes	yes	yes	no	no	no	tree
Pandorabots [18]	yes	no	yes	yes	no	yes	yes	no	DSL
Rasa [21]	yes	yes	no	yes	yes	yes	yes	yes	tree
SmartLoop [24]	yes	yes	yes	yes	yes	yes	yes	no	context
Watson [28]	yes	yes	yes	yes	yes	yes	yes	yes	context
Xatkit [8]	yes	yes	yes	yes	yes	yes	yes	yes	context
Xenioo [31]	no	yes	yes	yes	yes	yes	yes	no	tree

ask for a value when the parameter is required but the user utterance does not include its value. Parameters are typed by entities (6^{th} column in the table). Our meta-model supports both predefined entities (enumeration *PredefinedEntity* with values *text*, *date*, *number*, *float* and *time*) and chatbot-specific ones (class *Entity*).

Chatbot-specific entities can be *Simple* entities, defined as a list of words with their synonyms, or *Composite* entities, made of other entities and text. For example, in our bakery example, we may define simple entities for the products (cake, cupcake, biscuit...) and flavours (chocolate, strawberry, vanilla...), and a composite entity combining both (⟨product⟩ with ⟨flavour⟩ flavour, ⟨flavour⟩ ⟨product⟩, ⟨flavour⟩ flavoured ⟨product⟩...).

Chatbots can perform different *Action*s. The most common ones are the following (see 7^{th} to 9^{th} columns in Table 1): sending a *Text* response to the user, which requires specifying the actual text for each chatbot language; sending an *Image* which is identified by its *URL*; performing an *HttpRequest* to a given *URL*, optionally providing some *headers* and *data*; and sending to the user an *HttpResponse* for a previous *http* request.

Finally, a chatbot can define conversation *Flow*s. As the last column of Table 1 shows, all approaches provide some way to structure the dialogue, and in particular, the meta-model has primitives to cover conversation trees and intent activation based on contexts and sessions. Pandorabots supports a richer mechanism based on a DSL – the Artificial Intelligence Markup Language (AIML)[1] – which our meta-model does not include due to its specificity. A flow is made of *UserInteraction*s associated to an intent, and *BotInteraction*s comprising one or more actions. A flow must start with a user interaction followed by a bot interaction, after which there may be other user interactions, and so on.

To facilitate the instantiation of this meta-model, we have designed a textual concrete syntax for it. Listing 1 illustrates its usage by showing an excerpt of the definition of a chatbot for a bakery to which users can consult prices and order different products like bread or cakes. The first line defines the chatbot name and the supported languages (English and Spanish). Lines 4–18 define an intent named *Price*, which declares a set of training phrases for each language of the chatbot. If a set of phrases does not specify a language (as is the case in line 5), then they are assumed to be in the first language declared by the chatbot (English in this example). The intent defines four parameters in lines 15–18. The training phrases can refer to them (e.g., [count_param] in line 6) and assign them a value in the context of the phrase (e.g., three in line 6). The parameters type can be a predefined entity, like *number*, or a user-defined one, like *flavour*.

Lines 21–29 show the definition of the simple entity *flavour*. This declares the admissible flavours for each language supported by the chatbot, together with their synonyms.

Lines 31–42 illustrate the definition of actions, specifically, a text response called *PriceResponse*. As in the training phrases, text responses can be in different languages, and use parameter values (e.g., [Price.bread_param] in line 34).

Finally, lines 44–49 define the conversation flow (i.e., sequences of user and chatbot interactions). The listing configures two flows, which always must start with a user interaction and the corresponding intent. Flows are defined once, independently of the language. The flow in line 45 takes place when the user utterance matches the *Price* intent, in which case, the chatbot performs the action *PriceResponse* defined in lines 32–42. The second flow (lines 46–49) corresponds to the intent *Buy*. In this case, the chatbot asks for the product type to buy, and the flow is split depending on the user answer (cake or bread). This branching can be recursively nested to enable a compact representation of alternative flows.

The DSL includes model validation rules of two kinds. The first ones are *integrity constraints* that ensure the well-formedness of chatbot models. For ex-

[1] `http://www.aiml.foundation/`

ample, some of these rules forbid equally named elements (e.g., two *Action*s with the same name) and validate that each *Intent* has exactly one *LanguageIntent* for each language of the chatbot (attribute *Chatbot.lang*). The second kind of rules performs a static analysis of the chatbot definition to assess whether it adheres to *best practices* for chatbot design. Violating these rules may be a "smell" of a bad chatbot design. Currently, the DSL validates the following aspects: there is a fallback intent; text responses only use parameters of intents appearing in the conversation flow; there are no two intents with the same training phrase; all intents define either one regular expression or at least three training phrases; and training phrases do not start by a parameter typed by the predefined entity *text*, as this would match any user utterance which can be problematic.

```
1  chatbot Bakery language: en, es
2
3  intents:
4    Price:
5      inputs {
6        "How much are" (three)[count_param] (bread)[bread_param] "?",
7        "How much is a" (cake)[cake_param] "?",
8        "How much is a" (chocolate)[flavour_param](cake)[cake_param] "?"
9      }
10     inputs in es {
11       "¿Cuanto cuesta el" (pan)[bread_param] "?",
12       "¿Cuanto cuesta una" (tarta)[cake_param] "?",
13       "¿Cuanto cuesta un" (pastel)[cake_param] "de" (chocolate)[flavour_param] "?"
14     }
15     parameters:
16       bread_param, cake_param: entity product;
17       flavour_param: entity flavour;
18       count_param: entity number;
19
20 entities:
21   simple entity flavour:
22     inputs in en {
23       chocolate synonyms choco, cocoa, truffle;
24       ...
25     }
26     inputs in es {
27       chocolate synonyms choco, cacao, trufa;
28       ...
29     }
30
31 actions:
32   text response PriceResponse:
33     inputs {
34       "The" [Price.bread_param] "costs 1 euro per unit",
35       "The" [Price.flavour_param] [Price.cake_param] "costs 10 euro per unit",
36       "The" [Price.cake_param] "costs 10 euro per unit"
37     }
38     inputs in es {
39       "El" [Price.bread_param] "cuesta 1 euro por unidad",
40       "Las" [Price.cake_param] "de" [Price.flavour_param] "cuestan 10 euros por unidad",
41       "Las" [Price.cake_param] "cuestan 10 euros por unidad"
42     }
43
44 flows:
45   – user Price => chatbot PriceResponse;
46   – user Buy => chatbot Type {
47     => user Cake => chatbot Quantity => user num => chatbot BuyCakeHttp, buyCakeResponse;
48     => user Bread => chatbot Quantity => user num => chatbot BuyBreadHttp, buyBreadResponse;
49   }
```

Listing 1: Excerpt of chatbot model definition with the Conga DSL.

5 Recommending a chatbot creation tool

Due to the large amount of tools and approaches for chatbot creation (cf. Table 1), selecting the best option to build a particular chatbot becomes complex. To assist in this task, we provide a recommender that receives a chatbot model specified with CONGA and the answers to a questionnaire relative to other aspects of the chatbot (e.g., technical, organizational or managerial requirements), and from this information, it recommends an appropriate tool to implement the chatbot. The recommender builds on a model-based extensible architecture that enables the addition of new chatbot creation tools and the customization of the questions and model features the recommendation builds on.

Fig. 4 shows the meta-model our *Recommender* relies on. To make a recommendation, it considers a list of chatbot *Requirements*, whose value can be retrieved either by means of a *Question* to the developer, or automatically via an *Analysis* of the chatbot model. Both kinds of requirements have a *name*, a *text*, a list of admissible *Option*s, and can be *multi-response* or not. In addition, *Analysis* requirements define an *evaluator*, which is the (Java) class in charge of analysing the chatbot model. This latter class must extend the built-in abstract class *Evaluator* and implement its abstract method *evaluate*, which receives a chatbot model and returns the *Option*s that this model fulfils. The recommendation consists of a list of *Tools*. For each tool, the recommender stores the requirement options that are *available*, *unavailable*, *unknown* or are ultimately *possible* (i.e., not natively supported but achievable using a workaround).

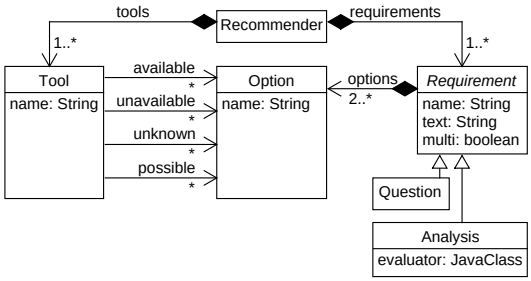

Fig. 4: Recommender meta-model.

The recommender currently considers the requirements in Table 2, and new ones can be added if needed. The table also shows the coverage of these requirements by two chatbot creation tools: Dialogflow and Rasa. Regarding analysis requirements, we check whether the chatbot model is multi-language (like in Listing 1), the targeted languages[2], and whether it uses predefined or chatbot-specific entities, calls to external services, parameters, training phrases or regular expressions. Rasa does not support multi-language bots, but a workaround is generating one bot per language, hence the value *possible* in the table.

Questions are chatbot requirements explicitly asked to the developer as they cannot be inferred from the chatbot model. The first seven questions in Table 2 deal with technical aspects. Specifically, we ask for the following issues: the social network the chatbot is to be deployed in (Dialogflow supports 16, and Rasa 8); the hosting server of the chatbot, since some platforms (e.g., Dialogflow) can host the chatbot themselves, but others (e.g., Rasa) require an external server; the level of support for version control, which is built-in in platforms like Di-

[2] For brevity, Table 2 shows the number of languages supported, not the list of them.

Table 2: Requirements that the recommender currently takes into consideration.

Text	Multi-response	Options	Dialogflow	Rasa
Analyses				
Is the chatbot multi-language?	false	Yes	avail.	possib.
		No	avail.	avail.
Which are the chatbot languages?	true	-	21	all
Does the chatbot use new or predefined entities?	true	Predefined	avail.	avail.
		New entities	avail.	avail.
		None	avail.	avail.
Does the chatbot call to external services?	false	One	avail.	avail.
		Multiple	possib.	avail.
		None	avail.	avail.
Does the chatbot use phrase parameters?	false	Yes	avail.	avail.
		No	avail.	avail.
Does the chatbot need persistent or volatile parameter storage?	true	Persistent	avail.	avail.
		Volatile	avail.	avail.
		None	avail.	avail.
Does your chatbot need natural language processing or pattern matching?	true	NLP	avail.	avail.
		Pattern	unavail.	unavail.
Questions				
Which social networks do you want to deploy the chatbot in?	true	-	16	8
Do you want to deploy the chatbot on your own host?	false	Tool host	avail.	unavail.
		Own host	unavail.	avail.
Do you want to use a built-in version control system?	false	Yes	avail.	avail.
		No	avail.	avail.
Do you require native support for chatbot analytics?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require native support for utterance persistence?	false	Yes	avail.	avail.
		No	avail.	avail.
Do you require the chatbot to support speech recognition?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require the chatbot to support sentiment analysis?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require to use an open-source tool?	false	Yes	unavail.	avail.
		No	avail.	avail.
Which price model do you plan to use?	true	Free	avail.	avail.
		Pay as you go	avail.	unavail.
		Quota	unavail.	unavail.
		Pay advanced feats.	unavail.	avail.
What's the level of expertise of the development team?	false	Low	avail.	unavail.
		High	avail.	avail.

alogflow, while programming-based approaches like Rasa need to use an external version control system like *github*; the need to monitor the chatbot performance (e.g., Dialogflow provides some chatbot analytics); the persistence of utterances for their subsequent analysis; and the need to support speech recognition or sentiment analysis.

The last three questions in Table 2 tackle organizational and managerial aspects concerned with open-source and price model requirements, and the level of expertise of the development team. For example, the expertise for using Rasa is higher than for Dialogflow, since the former requires programming.

Since some requirements may be more important than others depending on the project, we assign an importance level to each requirement, which the developer can customize. The supported levels are: *irrelevant*, *relevant*, *double relevant* and *critical*. Irrelevant requirements are not considered for the recommendation,

and critical ones are breaking factors (i.e., tools that do not comply with the requirement will not be recommended). For each tool, the recommender computes a score based on the supported requirements and their importance level. *Available* requirements add 1 to the score of a tool, *unavailable* ones add 0, *unknown* ones add 0.5, and *possible* ones add 0.75. In all cases, double relevant requirements score double. Then, the recommender orders the tools according to their score, and produces a report with the ranking of tools and how each requirement contributes to this ranking.

Incorporating a new chatbot creation tool (e.g., Watson) into our framework requires: (i) informing the tool options for every requirement in the recommender; (ii) providing a code generator from CONGA to the tool; (iii) optionally, providing a parser if reverse engineering is required. Our framework prevents the code generation for a tool whenever the chatbot requirements are *unavailable* in that tool. There may be some *possible* requirements though, meaning that their support is not native in the tool but they can be implemented. For instance, Rasa does not support multi-language chatbots, but this can be emulated by generating one chatbot per language. As another example, Dialogflow only supports one external service call per intent, and so, the generator only considers the first call and warns the developer.

6 Tool support

We have built tool support for our approach. Fig. 5(a) shows the developed editor for the CONGA DSL, which uses the Eclipse Modeling Framework (EMF) [25] and Xtext. The editor provides syntax highlighting, autocompletion, and informs of errors and warnings found in the chatbot models.

Upon uploading a chatbot model to a web server, we can apply the recommender (Fig. 5(b)) and generate code for a specific chatbot creation tool. Currently, the recommender considers 14 up-to-date tools, and we provide generators and parsers from/to Dialogflow and Rasa. Anyhow, as previously explained, both aspects are extensible. Figs. 5(c.1) and 5(c.2) show two generated chatbots for Dialogflow and Rasa in their respective development environments, from where the chatbots can be deployed into a social network.

7 Evaluation

This section reports on an evaluation of our approach on a migration scenario which involves both backward and forward engineering. The goal is to answer two research questions (RQs): **RQ1**: *Is* CONGA *expressive enough to capture the details of existing chatbots?* **RQ2**: *Can the migration process be fully automated?* For this purpose, we have migrated four Dialogflow agents developed by third parties (three from github, one built by Google) into Rasa. Table 3 summarizes the experiment results.

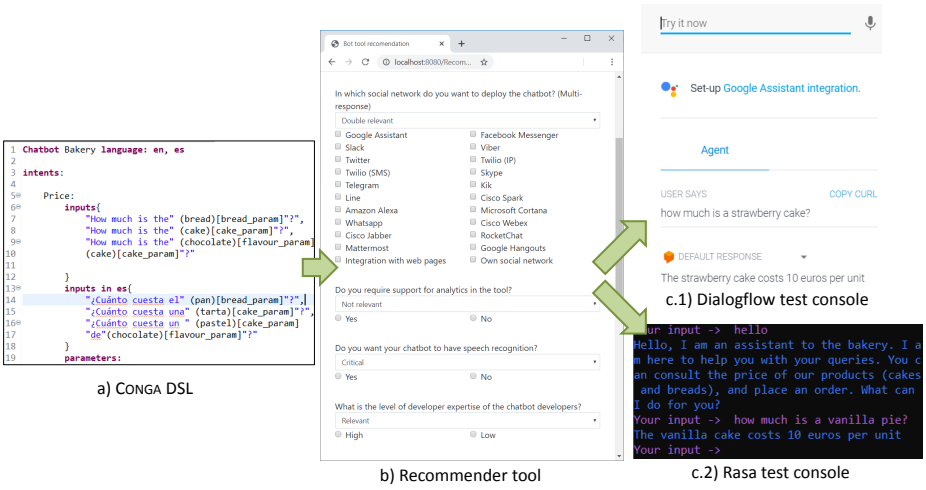

Fig. 5: Our tool in action for forward engineering. (a) CONGA editor. (b) Recommender. (c.1) Generated bot for Dialogflow. (c.2) Generated bot for Rasa.

Table 3: Assessment metrics.

	Dialogflow					CONGA		Rasa			
	No. intents	No. ents.	Http req.	No. files	Lang	No. objects	No. lines	No. chatbots	No. Python lines	No. Markd. lines	No. yaml lines
Game	11	0	yes	30	en/fr	541	268	2	378	242	362
Room reservation	7	1	no	17	en	717	196	1	253	166	137
Coffee shop	21	8	no	60	en	931	393	1	657	394	269
Nutrition	4	7	no	23	en	833	610	1	802	81	99

Game[3] is a conversational agent for a numeric guessing game. It has 11 intents, no entities, one *http* request, and supports English and French. Its Dialogflow specification is made of 30 JSON files. From this specification, our parser creates a model with 541 objects and 268 lines of CONGA code. Since Rasa does not support multi-language chatbots, two Rasa chatbots are generated from the CONGA model, one for each language. These have 378 lines of Python code (to define parameters and actions), 242 lines of Markdown code (to define intents and flows) and 362 lines of YAML code (to configure the chatbot).

Room reservation[4] is a chatbot to book hotel rooms. It has 7 intents and one entity, and works in English. The migration produces a Rasa chatbot with 253 lines of Python code. Since the original Dialogflow chatbot has button actions, which are unsupported by CONGA, we had to add them manually in Rasa.

Coffee shop is a Dialogflow pre-built agent to order food to a coffee shop. Its specification is the most complex of the four chatbots, spanning 60 JSON files. These are parsed into a CONGA model with 931 objects.

[3] `https://github.com/actions-on-google/dialogflow-number-genie-nodejs`

[4] `https://github.com/dialogflow/dialogflow-java-client-v2/tree/master/samples/resources`

Nutrition[5] is a chatbot to query the nutritional value of meals. Although it is a small chatbot with 4 intents and 7 entities, it generates many lines of Python code because the entities have many entries.

Overall, we were able to migrate all Dialogflow chatbots but the button actions on the room reservation bot, which confirms the expressiveness of Conga (RQ1). Except for that bot, migration was fully automatic (RQ2). These results are very promising, but more case studies are needed to strengthen the confidence in the capabilities of Conga. Moreover, we manually checked that the produced Rasa chatbots preserved the original Dialogflow behaviour, but we plan to automate this check in future work (e.g., using tools like Botium[6]).

8 Related work

The popularity of chatbots has promoted the appearance of many tools for their construction. In this section, we revise works built atop these tools to simplify some aspect of chatbot development.

Xatkit [8] (formerly known as Jarvis [7]) is a model-driven solution for developing chatbots. Similar to our approach, it proposes a meta-model and a textual DSL. However, differently from us, Xatkit has its own bot execution engine that builds on Dialogflow to identify the user intent using NLP, and does not generate code for existing chatbot development tools. Moreover, even though Xatkit is model-based, it does not address the recommendation of suitable chatbot platforms, nor reduces the risk of vendor lock-in by supporting chatbot migration.

In [3], Baudat et al. facilitate the definition of Watson chatbots by means of an OCaml library which produces the necessary JSON files, and the use of ReactiveML to orchestrate the dialog. While this approach is generative, it is limited to Watson and does not support reverse engineering.

There are some recent model-based proposals to automate the construction of chatbots for a specific task. For example, the framework in [1] permits creating chatbots for video game development; in [20], we generate Dialogflow chatbots to allow instantiating meta-models using a NL syntax; and in [19], we generate model query chatbots. Other works do not rely on models for automating chatbot creation, such as [13], where the authors enable a black-box reuse of components for creating chatbots for FAQ exploration. All these approaches are not general-purpose, but they produce chatbots for a specific task (creating video games, creating models, querying models, or exploring FAQs).

Conversely, in [2], the authors envision a reverse engineering process called *botification* to produce a conversational interface for existing web sites. The process parses a web page to produce a domain model, which serves to configure the allowed NL interactions. Botified webs improve the user experience for visually impaired users, and the development cost is low. We believe that our architecture could serve as a reference to implement this scenario.

[5] `https://github.com/Viber/apiai-nutrition-sample`
[6] `https://www.botium.ai/`

Another related line of research concerns crowd-powered conversational assistants [11, 12]. While they are not auto-generated, as we do in this paper, they can auto-evolve by learning appropriate responses from previous ones.

Finally, some development tools are specific for voice-user interfaces. For example, tortu[7] supports the visual creation of conversation flows, but it does not allow code generation or bot migration. In a similar vein, VoiceFlow[8] offers a graphical DSL to create voice-based conversation flows that can be deployed on Google home or Alexa, but does not provide recommendation or migration facilities, and the deployment platforms are fixed.

Overall, our approach is novel as it provides a complete MDE solution comprising a unifying DSL for chatbot design, a recommender of up-to-date chatbot development tools according to given design and technical chatbot requirements, and supporting forward and backward engineering, including migration.

9 Conclusion and future work

Nowadays, we can find many tools for building chatbots. While these tools accelerate chatbot development, the chatbot design can become obscured under technical tool details. Moreover, selecting the most appropriate tool, or chatbot migration, require a high investment of time. To alleviate these problems, we have proposed an MDE approach to chatbot development that includes a textual DSL, a platform recommender, code generators and parsers. Our approach supports both forward and reverse chatbot engineering, and has been evaluated by migrating four Dialogflow chatbots developed by third parties to Rasa.

In the future, we plan to extend our framework with more chatbot creation tools, facilities for model-based testing, quick-fixes for violations of chatbot best-practices, and mechanisms to make Conga extensible with platform-specific concepts, like buttons. We are currently migrating our editor of Conga models to a web environment, and later we plan to perform a user study with developers to assess the advantages of our approach. Finally, we plan to create higher-level DSLs to define domain-specific chatbots (e.g., for education or commerce) which can be transformed into our framework for validation and code generation.

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

References

1. Baena-Pérez, R., Ruiz-Rube, I., Dodero, J.M., Bolivar, M.A.: A framework to create conversational agents for the development of video games by end-users. In: OLA. CCIS, vol. 1173, pp. 216–226. Springer (2020)
2. Báez, M., Daniel, F., Casati, F.: Conversational web interaction: Proposal of a dialog-based natural language interaction paradigm for the web. In: CONVERSATIONS. LNCS, vol. 11970, pp. 94–110. Springer (2019)

[7] `https://tortu.io/` [8] `https://www.voiceflow.com/`

3. Baudart, G., Hirzel, M., Mandel, L., Shinnar, A., Siméon, J.: Reactive chatbot programming. In: REBLS@SPLASH. p. 2130. ACM (2018)
4. Botkit: `https://botkit.ai/`, last access in 2020
5. Chatfuel: `https://chatfuel.com/`, last access in 2020
6. Chatterbot: `https://chatterbot.readthedocs.io/`, last access in 2020
7. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Multi-platform chatbot modeling and deployment with the Jarvis framework. In: Proc. CAiSE. LNCS, vol. 11483, pp. 177–193. Springer (2019)
8. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: A multimodal low-code chatbot development framework. IEEE Access **8**, 15332–15346 (2020)
9. Dialogflow: `https://dialogflow.com/`, last access in 2020
10. FlowXO: `https://flowxo.com/`, last access in 2020
11. Huang, T.K., Chang, J.C., Swaminathan, S., Bigham, J.P.: Evorus: A crowd-powered conversational assistant that automates itself over time. In: UIST. pp. 155–157. ACM (2017)
12. Jonell, P., Fallgren, P., Dogan, F.I., Lopes, J., Wennberg, U., Skantze, G.: Crowd-sourcing a self-evolving dialog graph. In: CUI. pp. 14:1–14:8. ACM (2019)
13. de Lacerda, A.R.T., Aguiar, C.S.R.: FLOSS FAQ chatbot project reuse: How to allow nonexperts to develop a chatbot. In: OpenSym. ACM (2019)
14. Landbot.io: `https://landbot.io/`, last access in 2020
15. Lex: `https://aws.amazon.com/en/lex/`, last access in 2020
16. LUIS: `https://www.luis.ai/`, last access in 2020
17. Microsoft Bot Framework: `https://dev.botframework.com/`, last access in 2020
18. Pandorabots: `https://home.pandorabots.com/`, last access in 2020
19. Pérez-Soler, S., Daniel, G., Cabot, J., Guerra, E., de Lara, J.: Towards automating the synthesis of chatbots for conversational model query. In: EMMSAD. LNCS (2020)
20. Pérez-Soler, S., González-Jiménez, M., Guerra, E., de Lara, J.: Towards conversational syntax for domain-specific languages using chatbots. Journal of Object Technology **18**(2) (2019)
21. Rasa: `https://rasa.com/`, last access in 2020
22. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006)
23. Shevat, A.: Designing bots: Creating conversational experiences. O'Reilly (2017)
24. SmartLoop: `https://smartloop.ai/`, last access in 2020
25. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework, 2nd edition. Pearson Education (2008)
26. Tegos, S., Demetriadis, S.N.: Conversational agents improve peer learning through building on prior knowledge. Educ. Technology & Society **20**(1), 99–111 (2017)
27. Väänänen, K., Hiltunen, A., Varsaluoma, J., Pietilä, I.: CivicBots - chatbots for supporting youth in societal participation. In: CONVERSATIONS. LNCS, vol. 11970, pp. 143–157. Springer (2019)
28. Watson: `https://www.ibm.com/cloud/watson-assistant/`, last access in 2020
29. Winkler, R., Hobert, S., Salovaara, A., Söllner, M., Leimeister, J.M.: Sara, the lecturer: Improving learning in online education with a scaffolding-based conversational agent. In: CHI. pp. 1–14. ACM (2020)
30. von Wolff, R.M., Nörtemann, J., Hobert, S., Schumann, M.: Chatbots for the information acquisition at universities - A student's view on the application area. In: CONVERSATIONS. LNCS, vol. 11970, pp. 231–244. Springer (2019)
31. Xenioo: `https://www.xenioo.com/en/`, last access in 2020