The Journal of Systems and Software 231 (2026) 112600

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A model-based solution for automated (Re-)engineering of task-oriented chatbots☆

Sara Pérez-Soler [a], Esther Guerra [b], Juan de Lara [b,*]

[a] Universidad Francisco de Vitoria, Madrid, Spain
[b] Modelling & Software Engineering Research Group Universidad Autonoma de Madrid, Madrid, Spain

ARTICLE INFO

Dataset link: https://saraperezsoler.github.io/CONGA/, http://dimo1.ii.uam.es/CONGA/, https://github.com/Conga-dsl/ValidationSetup, https://github.com/Conga-dsl/BotiumExperiment

Keywords:
Task-oriented chatbots
Model-driven engineering
Domain-specific languages
Migration
Recommendation system

ABSTRACT

Chatbots are popular to access all sorts of software services via natural language conversation. The increasing demand for task-oriented chatbots has triggered the proposal of many tools for their construction, like Dialogflow, Lex, Rasa, or Watson. However, selecting the most appropriate one is difficult; the conceptual design behind a chatbot may become buried under the tool technicalities; and migration between chatbot development platforms must be done manually.

To alleviate these problems, we propose a platform-independent design notation for task-oriented chatbots, based on the analysis of fifteen chatbot development platforms. Following model-driven engineering principles, the chatbot implementation is synthesised from the design, and designs can be extracted from the implementations, enabling the migration and re-engineering of chatbots. Moreover, a recommender suggests the most suitable platform for a given chatbot design, considering contextual factors.

We have realised these ideas in Conga: an extensible web application featuring a design notation editor; a development platform recommender; platform-specific validators; and generators and parsers for Dialogflow and Rasa. We evaluated Conga over 291 Dialogflow and Rasa open-source chatbots, showing its expressiveness, portability, and usefulness for finding chatbot quality issues (found in 93,8% of the chatbots). Overall, our architecture enables neutral chatbot designs, automates migration, and provides mechanisms for defect detection at the design level.

1. Introduction

Task-oriented chatbots offer a conversational interface for accessing software services (McTear, 2020). Their popularity is rising since they present some advantages over traditional software. Most notably, they enable interaction via natural language (appropriate for users with non-technical background), often through *channels* like social networks (which favours their use from mobile devices), and with different modalities (voice and text). Chatbots are widely used for customer support, to access leisure services (e.g., booking travel services or event tickets), for shopping, or as assistants for work activities (e.g., in software engineering tasks Santhanam et al., 2022 such as modelling Pérez-Soler et al., 2018; Ren et al., 2023, query refinement Zhang et al., 2022, access to API documentation Tian et al., 2017, understanding APIs Ed-Douibi et al., 2020, load testing Okanović et al., 2020, or smart contract development Qasse et al., 2021).

The growing interest in task-oriented chatbots has led to the emergence of many platforms, tools and libraries for their construction (Pérez-Soler et al., 2021). Many big software companies and cloud vendors have created their own solutions, like Google's Dialogflow (Dialogflow, 2024), IBM's Watson assistant (Watson, 2024), Microsoft Bot Framework (2024), or Amazon's Lex (Lex, 2024). In addition, many other choices for chatbot development exist, including frameworks like Rasa (2024), or Xatkit (Daniel et al., 2020); low-code platforms like Flow XO (2024), Landbot (2024), or Chatfuel (2024); solutions based on domain-specific languages (DSLs) like Pandorabots (2024); and libraries like ChatterBot (2024) and LUIS (2024).

This plethora of tools brings chatbot developers a wide range of possibilities, but makes it *difficult to decide which is the best tool* for a given scenario (**challenge 1**). Tools offer different support not only for technical aspects like natural language processing (NLP), dialogue

☆ Editor: Professor Uwe Zdun.

* Corresponding author.
E-mail addresses: Sara.Perez@ufv.es (S. Pérez-Soler), Esther.Guerra@uam.es (E. Guerra), Juan.deLara@uam.es (J. de Lara).

https://doi.org/10.1016/j.jss.2025.112600
Received 8 February 2024; Received in revised form 20 April 2025; Accepted 8 August 2025
Available online 25 August 2025

specification, deployment, integration with other systems, or testing; but also for managerial concerns like pricing model, code hosting, collaboration support, or software distribution model (open/closed source) (Pérez-Soler et al., 2021).

Moreover, the underlying *conceptual design of a chatbot implementation is hard to grasp* (**challenge 2**). This is due to the accidental complexity that the chosen platform imposes (e.g., the configuration and Python files of Rasa chatbots, or the various forms and scripts of Dialogflow chatbots). This heterogeneity also makes it *complex to express and assess quality assurance rules for chatbots across different platforms* early, in the design phase (Cuadrado et al., 2024a; Cañizares et al., 2024b) (**challenge 3**). As a consequence, quality issues may go unnoticed during development, affecting the final chatbot quality. To address this, early error detection mechanisms are needed, applicable even before the chatbot is deployed and fully functional (Cuadrado et al., 2024a), since the cost of fixing bugs increases as the development process progresses.

Finally, once deployed, *chatbots are often difficult to migrate to other platforms*, e.g., to modernise them, or to take advantage of novel features or pricing plans offered by these other platforms (**challenge 4**). This is especially important given the rapidly evolving ecosystem of chatbot construction tools and platforms, and highlights the need for mechanisms to automate migration and modernisation of existing chatbots, e.g., into generative artificial intelligence technologies (Cuadrado et al., 2024b). While current research targets the creation of new chatbot development tools, there is scarce support for automating chatbot migration, or for the design of chatbots in a technology-neutral way.

To address the identified challenges, we propose a model-driven solution (Brambilla et al., 2017) for chatbot design and re-engineering. Our solution is based on a neutral chatbot design language, called Conga, devised on the analysis of fifteen prominent chatbot development tools. Given a chatbot design and a characterisation of the managerial aspects that it should fulfil, a recommender system helps the chatbot developer select the most appropriate implementation tool, and code generators synthesise the chatbot implementation for the tool of choice. To support migration, our solution includes parsers from specific chatbot development tools into Conga. This way, the chatbot design can be reverse-engineered from the implementation, statically analysed to look for errors, improved if needed, and then migrated to another tool using the code generators. Our solution currently supports the complete re-engineering process for Dialogflow and Rasa chatbots, but is extensible to other tools. While the paper presents the whole Conga framework, it focuses on two specific usage scenarios: (i) expressing quality rules to detect issues in existing chatbots, and (ii) serving as an intermediate representation that facilitates migration between chatbot technologies. The use of Conga for direct engineering, and its evaluation with users or against other chatbot notations or tools, are left for future work.

This paper extends our preliminary paper at the ER conference (Pérez-Soler et al., 2020b), and the tool demo at the ICSE conference (Pérez-Soler et al., 2021), as follows. We have substantially expanded those works by a more elaborate chatbot design notation supporting complex conversations (e.g., loops) and richer chatbot outputs (e.g., buttons); an extensible architecture that enables adding new validators, parsers, generators and recommenders for specific chatbot development platforms; the implementation of validators, parsers and generators for Rasa and Dialogflow; the availability of a full-fledged, extensible, service-based web platform supporting the approach; and an extensive evaluation of Conga for the two use cases tackled in the paper: chatbot quality assurance and chatbot migration. This evaluation is performed over 291 open-source Dialogflow and Rasa chatbots, and aims to answer the following research questions (RQs):

RQ1 Can Conga help detecting quality issues in existing chatbots?
RQ2 Is Conga expressive enough to capture the details of Dialogflow and Rasa chatbots?
RQ3 To what extent does Conga automate the migration between Rasa and Dialogflow?

This way, the novel research contributions of our work are the following:

(i) A neutral notation called Conga to express chatbot designs. The notation is useful for quality assurance (by incorporating generic and technology-specific well-formedness rules for chatbot designs), direct chatbot engineering (i.e., synthesise chatbots for specific technologies), reverse chatbot engineering (i.e., extract the design underlying a chatbot implementation), and chatbot migration (by using Conga as a bridge language for translation between technologies). The focus of this paper is on quality assurance and chatbot migration.
(ii) A modular, extensible recommender for selecting the most suitable chatbot implementation tool based on the chatbot features and other practical concerns. Its model-based architecture enables the inclusion of emerging concerns and technologies.
(iii) A realising framework that features chatbot quality assurance, direct/reverse chatbot engineering, chatbot migration, and recommendation of chatbot tools.
(iv) Three empirical experiments over a dataset of 291 third-party chatbots — to our knowledge, the largest chatbot dataset reported in the literature. The experiments show the feasibility of automated chatbot migration via Conga, and the capabilities of Conga to detect quality issues in real chatbots.

In the remainder of this paper, first, Section 2 introduces the main concepts of chatbots. Then, Section 3 overviews our proposal, and the following three sections give details on the Conga DSL (Section 4); the validation, parsing and generation services (Section 5); and the recommender of chatbot development platforms (Section 6). Section 7 presents our tool support. Next, Section 8 reports on the evaluation of our approach. Finally, Section 9 compares with related research, and Section 10 finishes with the conclusions and open research lines. The Appendix provides full details on the parsers and generators from/to Conga and Dialogflow and Rasa.

2. Background on chatbots

Chatbots are conversational systems with a natural language interface. They can work in open-domains, as is the case of ChatGPT (OpenAI, 2024), or be oriented to solve particular tasks. The latter are the focus of our work.

Task-oriented chatbots are agents enabling the access to software services via conversation. They can be deployed in a variety of channels, including social networks, web sites or smart speakers. Combining these two aspects – interaction in natural language and flexibility of use – has made them a successful paradigm nowadays.

In general, open-domain chatbots are based on generative artificial intelligence, in particular on large language models (LLMs) (Zhao et al., 2023). These could be fine-tuned for a particular task, or be instructed to accomplish a task by means of prompts. However, the technology to achieve reliable, robust, trustworthy task-oriented chatbots using LLMs is still in the making (Zamfirescu-Pereira et al., 2023). For this reason, instead, task-oriented chatbots are typically designed around a set of user *intents*, or conversation topics, that the chatbot aims at recognising. Fig. 1 depicts their working scheme. Whenever the user says an *utterance* in natural language to the chatbot (label 1 in the figure), the chatbot matches the most likely intent for the phrase (label 2). If no intent matches, there is usually a *fallback* intent that the

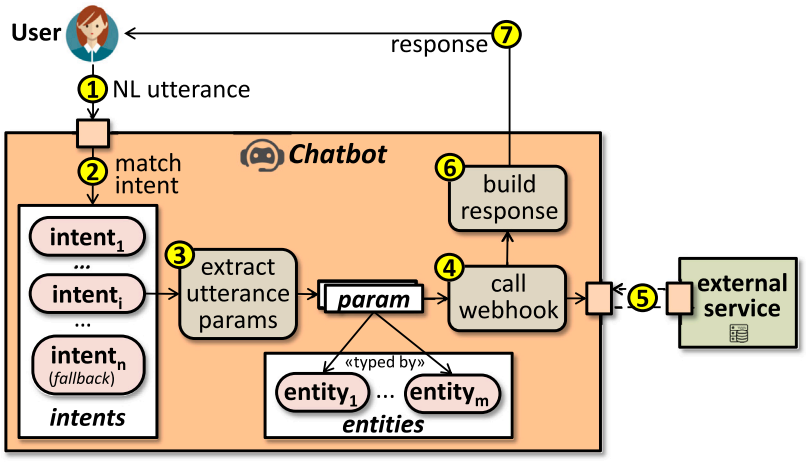

Fig. 1. Chatbot working scheme.

chatbot employs to ask the user for a clarification. For example, a chatbot for a pizzeria may have two main intents (one to allow users to order pizza, and another to inform about the types of pizza served), plus a greeting intent, and a fallback intent.

Once an intent has been matched, the chatbot may require extracting *parameters* from the user utterance (label 3). In the pizzeria example, the user may say *"I'd like a big Hawaiian pizza"*, which the chatbot would match to the *order pizza* intent, extracting the type (*Hawaiian*) and size (*big*) of the pizza. Parameters are typed by *entities*, which can be either predefined (e.g., numbers, dates) or user-defined (e.g., pizza sizes, pizza types), and can be tagged as optional or mandatory. If an utterance does not include the value of mandatory parameters, then the chatbot prompts the user for their value.

Upon obtaining the required information, the chatbot may need to access an external service via a *webhook* (labels 4 and 5) to manage the user intent. In the pizzeria example, this would be the shop information system that the chatbot uses to store the order and obtain its identifier and price. At this point, the chatbot is ready to answer the user (labels 6 and 7). Responses normally include text, but may also contain other elements such as images, links, or widgets specific to the deployment channel, like buttons in Telegram.

Finally, chatbots may have predefined conversation flows consisting of the interleaving of expected intents and chatbot response actions, in user and chatbot turns.

Fig. 2 exemplifies a possible conversation for the pizzeria chatbot. When the user starts the conversation, the chatbot detects the *greeting* intent and replies by asking the user what he or she wants. Then, the user asks for a pizza, and the chatbot recognises the *order pizza* intent, but it misses the mandatory parameter *pizza size*, so it prompts for its value. The user provides the size, and the chatbot responds asking for further items to order. The user does not want anything else, so the chatbot proceeds to checkout (omitted in the figure).

Overall, to build a chatbot like the pizza-bot, developers must specify its elements (intents, entities, conversation flows, etc.) in the chatbot implementation tool of choice. However, each tool has its own specification formats and may have different technical and managerial capabilities. Hence, the choice of tool is of great importance, as it can determine some features of the chatbot, which, once implemented, may be difficult to migrate to another technology. Furthermore, having the chatbot directly implemented in a tool makes it difficult to understand the underlying design of the chatbot, reasoning about this design, and detecting quality issues at the design level (Cuadrado et al., 2024a).

3. Overview of the approach

Our approach permits modelling task-oriented chatbots independently of the chatbot development tool, to enable forward and backward engineering of chatbot code from the designed chatbot models.

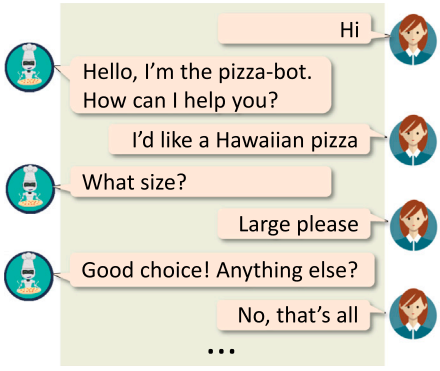

Fig. 2. Example of conversation with a chatbot for a pizzeria.

This enables reasoning and validation of chatbot models prior to their implementation, keeping the chatbot design independent of a specific technology. Moreover, it facilitates coping with a variety of scenarios, including forward engineering of chatbots, chatbot evolution, chatbot maintenance, and chatbot migration between technologies and between different versions of a same technology.

Fig. 3 shows a scheme of our proposal. The main component is a web platform where chatbot developers can design chatbots using a technology-agnostic DSL called Conga (ChatbOt modelliNg lanGuAge). The DSL is built based on a neutral meta-model resulting from an analysis of fifteen of the most prominent task-oriented chatbot development tools used nowadays, reported in Pérez-Soler et al. (2021). The DSL permits defining chatbot models independently of any development platform, and will be described in Section 4.

The DSL is complemented with code generators that synthesise code from the chatbot models into specific development tools (e.g., JSON files in the case of Dialogflow, or Python, markdown and YAML files in the case of Rasa). The generated chatbots can be deployed in a variety of platforms (e.g., Telegram, Slack or Twitter) to make them available to chatbot users. The DSL also facilitates chatbot migration by the provision of parsers from several development tools into the DSL. In addition, chatbot models can be validated, checking general chatbot quality criteria and well-formedness rules, as well as specific validations for particular platforms. Section 5 explains the parsing, generation and validation services, detailing the algorithms to generate, validate and parse code for Dialogflow and Rasa.

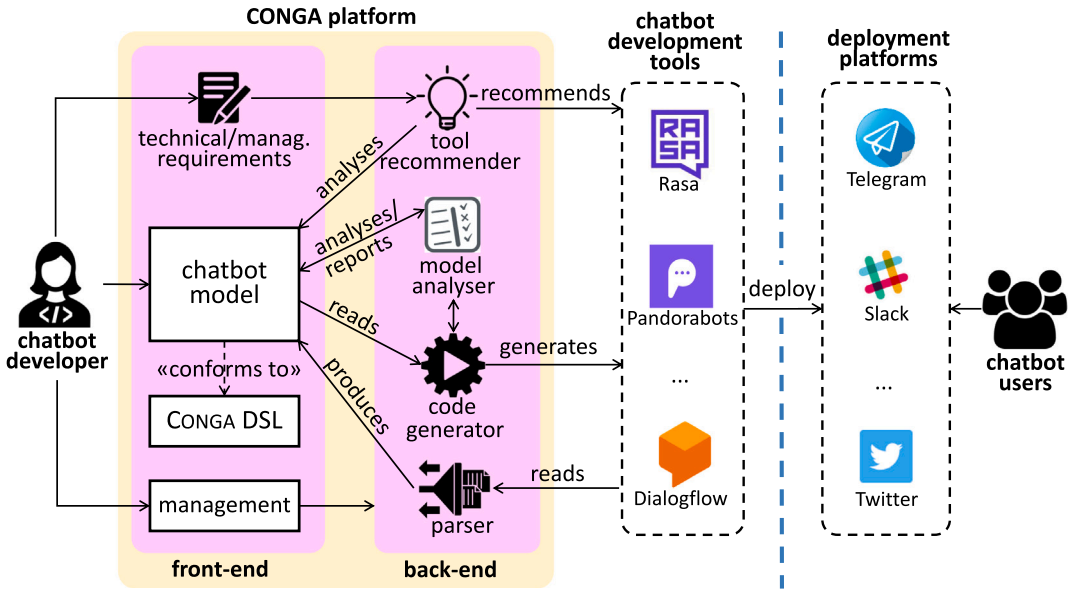

Fig. 3. Schema of our approach.

To facilitate the task of selecting a development tool for implementing a given chatbot model, our platform incorporates an extensible recommender that analyses the chatbot model and other technical requirements to provide a ranked list of suitable tools. Section 6 introduces the recommender system and its extensible architecture.

Finally, the CONGA platform incorporates a management module, by which chatbot developers can extend the platform with new code generators, parsers and specific validations for other chatbot development tools, or new versions of the tools already supported. Section 7 will describe the architectural decisions that enable this extensibility.

Overall, the goal of CONGA is not to become a competing notation to the existing ones, as it does not provide an execution platform. Instead, it is a chatbot design notation intended to bridge chatbot technologies to facilitate reengineering existing chatbots (translating from one technology to CONGA, and then to another technology). CONGA also seeks to streamline the direct engineering of chatbots by decoupling design from implementation, supporting the validation of chatbots designs, and allowing the selection of the implementation technology to be deferred to a later phase, aided by a recommendation system. Finally, its extensible, service-based architecture allows the seamless integration of new chatbot technologies that may emerge. In this paper, the focus is on the use of CONGA for chatbot quality assurance and chatbot migration, features that are evaluated in Section 8.

4. The CONGA DSL

This section presents the abstract syntax of CONGA given by a meta-model (Section 4.1), and describes its textual concrete syntax through an example (Section 4.2).

4.1. Abstract syntax

In Pérez-Soler et al. (2020b), we analysed fifteen chatbot development tools and proposed an initial neutral meta-model to define chatbots. Fig. 4 shows an extended version of this meta-model, which is organised into four packages that handle the definition of *intents* (i.e., the user intention in the communication), *entities* (i.e., the domain concepts), *actions* (i.e., the chatbot reaction to user utterances), and *flows* (i.e., the user and chatbot conversation turns).

More in detail, the *intents* package contains the concepts related to the intents definition. A Chatbot has a name, one or more supported languages (which allows defining multi-language chatbots), and a list of intents. Intents have a name, can be fallback, and may define a set of TrainingPhrases per supported language. Training phrases are examples of how a user may express an intention. For instance, a training phrase to express the intention to order a pizza can be "*I want a medium thin crust pizza*". Intents can also define Parameters, which are pieces of information that the chatbot needs to extract from the user utterances. In the previous example, the chatbot needs to extract *medium* as the pizza size, and *thin crust* as the dough type, to be able to manage the order. Parameters have a name, a type, can be multivalued (a list), can be required, and can define a list of prompts to ask for a value when the parameter is required but the user utterance does not include its value. Parameter types can be either predefined (enumeration PredefEntity with entries text, date, number, float and time) or domain-specific (class Entity).

The *entities* package permits defining Entities. These have a name and a set of entries per language. Each Entry can be Simple, consisting of a word and its synonyms; Composite, made of other entities and literals; or a RegularExpression. For example, our pizza-bot may define simple entities for the size (small, medium, large) and kind (Veggie, Cheese, Pepperoni...) of pizzas; a composite entity for ordered products (⟨pizzaSize⟩ ⟨pizzaKind⟩ pizza); and a regular expression for the zip code in the delivery address.

The *actions* package defines six types of actions that a chatbot can perform: sending a Text response, which may refer to parameters (e.g., "*Small cheese pizza order confirmed*"); sending an Image identified by its URL; showing Buttons (e.g., to present each pizza ingredient as a button, hence facilitating the client's selection); performing an HttpRequest to an URL; and sending an HttpResponse for a previous HTTP request to the user (e.g., to place an order to the pizza store and send a confirmation message). The last action type, Empty, is a wildcard for other platform-specific actions (e.g., Python code in Rasa). These actions stem from the analysis of native actions that the 15 platforms studied in Pérez-Soler et al. (2021) support. Additionally, arbitrary code can be placed behind service endpoints and then be invoked using the HttpRequest/HttpResponse mechanism.

Finally, the *flows* package supports the definition of the conversation structure. The conversation alternates between user and chatbot turns (classes UserInteraction and BotInteraction). The user turns refer to an intent, and the chatbot turns define a list of actions that the chatbot has to perform. In addition, both UserInteraction and BotInteraction may have a label, which is useful to define conversation loops (e.g., to allow the pizza-bot to iteratively ask the user for more toppings to add to

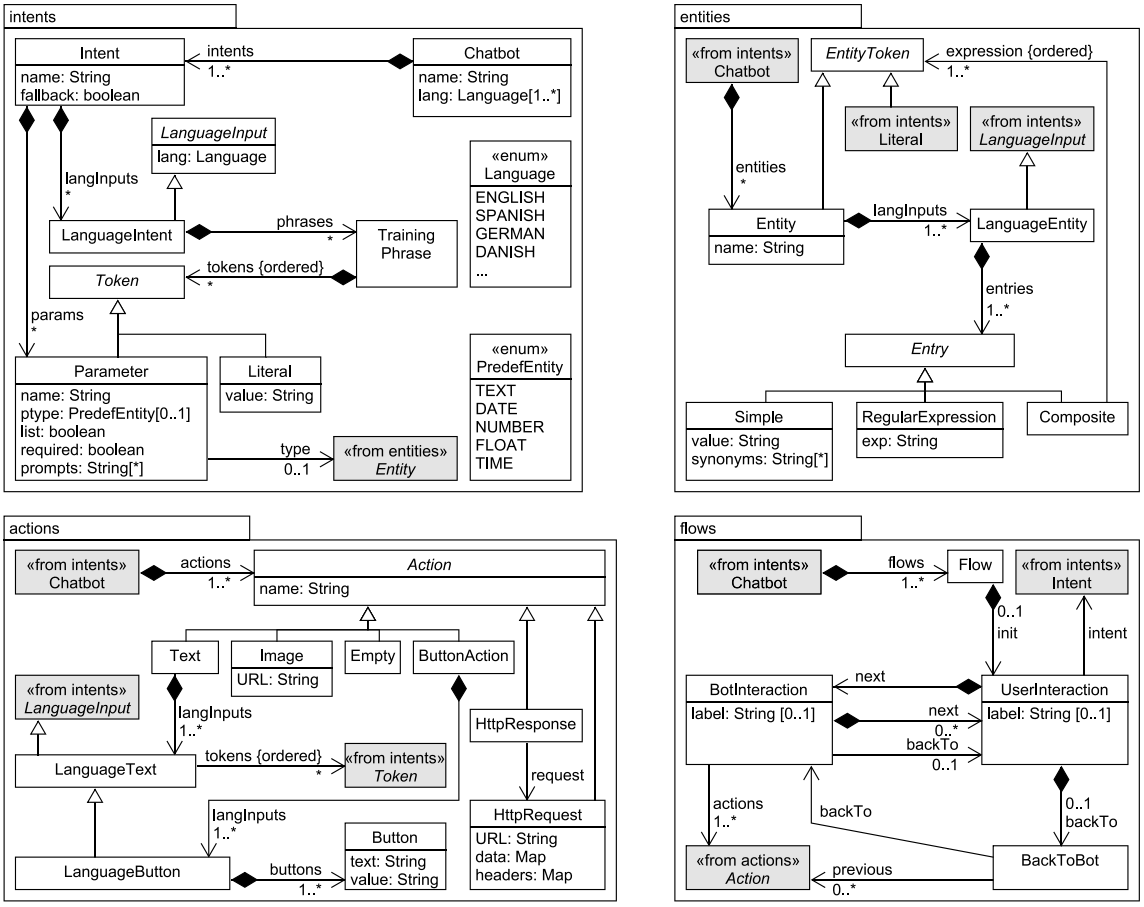

Fig. 4. Meta-model for chatbot design (extension of Pérez-Soler et al. (2020b)).

the ordered pizza). Specifically, the labels allow identifying the next interaction of a conversation loop. In addition, it is possible to go from a bot interaction back to a previous user interaction (reference backTo), and vice versa (class BackToBot, which may declare a list of Actions to perform previous to the jump). Again, we incorporated these flow constructs into Conga because all are native in some of the 15 platforms analysed in Pérez-Soler et al. (2021). Since conditionals (i.e., flow bifurcations), loops, and dynamic memory (i.e., parameters) make a language Turing-complete, we deem them sufficient to express chatbot dialogues.

Altogether, the meta-model of Conga provides the necessary primitives to define the chatbot behaviour, but excludes runtime services for chatbots like conversation persistence, analytics, or quality feedback. Such runtime services do not inherently influence the chatbot's behaviour or require consideration during its design. Instead, they can be plugged or unplugged after the chatbot has been deployed. In the future, we plan to develop libraries for such runtime services that designers may inject into the designed chatbots.

We argue that existing chatbot development platforms cannot play the role of a chatbot design language like Conga. This is so as reasoning about a chatbot design requires abstracting away from the concrete syntax and accidental details in these platforms. For instance, Rasa chatbots are defined using markdown, YAML and Python, and Dialogflow agents use graphical forms and can be exported to JSON; this makes their analysis complex. This is why we created a meta-model for chatbot designs. Moreover, no development platform is a superset of the others, lacking concepts that others offer (Pérez-Soler et al., 2021). For example, Rasa lacks loops, and Dialogflow misses explicit flow declarations. In contrast, a comprehensive neutral notation offers flexibility to capture designs across diverse technologies. Actually, pivot meta-models – like that of Conga – are common in model-driven engineering to bridge technological spaces (e.g., workflows in Salado-Cid et al. (2024)).

Overall, compared to the initial meta-model version in Pérez-Soler et al. (2020b), this extended version is modularised in packages, supports regular expressions, provides better handling of entity tokens, considers buttons and empty (i.e., platform-specific) actions, and permits loops in conversation flows. Section 5.1 will explain the handling of loops and their mapping into/from Dialogflow.

4.2. Concrete syntax

We have designed a textual concrete syntax for the Conga DSL, based on the meta-model described in the previous section. Listing 1 shows an example of a Pizza Store chatbot defined in Conga. The definition starts with the name of the chatbot and the list of supported languages, in this case English (line 1).

The keyword intents (line 2) begins the intents declaration section. Every intent definition starts with the intent name and several training phrases. In the case of multi-language chatbots, each intent should provide training phrases for each language that the chatbot considers. The listing defines four intents: StartOrder to start ordering a pizza (lines 3–10), Toppings to add a topping (lines 11–17), EndOrder to finish an order (lines 18–22), and ToppingsInfo to obtain the available toppings (lines 23–27). Intents may also declare parameters. In the listing, StartOrder has the parameters size and address (lines 8–10), and

```
1  Chatbot "Pizza Store" language: en
2  intents:
3     StartOrder: {
4           "I want to order a pizza",
5           ("Big")[size] "pizza",
6           "One" ("small")[size] "pizza, please"
7        }
8        parameters:
9           size: entity PizzaSize, required, prompts ["What pizza size do you want?"];
10          address: entity text, required, prompts ["What is your address?"];
11    Toppings: {
12          "extra" ("cheese")[toppings],
13          "with" ("ham")[toppings],
14          ("bacon")[toppings]
15       }
16       parameters:
17          toppings: entity Ingredients, required;
18    EndOrder: {
19          "That's all",
20          "Nothing else",
21          "No more toppings, thank you"
22       }
23    ToppingsInfo: {
24          "What toppings do you have?",
25          "Which are the toppings?",
26          "What toppings can I choose?"
27       }
28 entities:
29    Simple entity PizzaSize: {
30       small synonyms tiny, little
31       medium synonyms regular, intermediate
32       big synonyms huge, large
33    }
34    Simple entity Ingredients: {cheese ham pepperoni bacon}
35 actions:
36    Button response askingForToppings: {
37       text: "What toppings do you want?"
38       buttons:
39          – value: "cheese"      – value: "ham"         – value: "pepperoni"
40          – value: "bacon"       – value: "That's all"
41    }
42    Request post orderPizza:  URL: "https://mypizzaStore.com/order";
43       data: "size" : ["StartOrder.size"], "address" : ["StartOrder.address"];
44       dataType: JSON;
45    Request post noteTopping: URL: "https://mypizzaStore.com/topping";
46       data: "address" : ["StartOrder.address"], "toppings": ["Toppings.toppings"];
47       dataType: JSON;
48    Text response info: {
49       "We have cheese, ham, pepperoni and bacon" }
50 flows:
51    – user ToppingsInfo => chatbot info;
52    – user StartOrder => ask: chatbot askingForToppings {
53       => user Toppings => chatbot noteTopping back to ask;
54       => user EndOrder => chatbot orderPizza;
55    };
```

Listing 1: Defining a chatbot for a pizzeria with Conga.

Toppings has the parameter toppings (lines 16–17). A parameter must declare its name, its entity type, optionally if it is required or a list, and prompts that the chatbot would ask the user in case of missing values. The entity type can be predefined, like text for the parameter address, or user-defined, like PizzaSize for the parameter size, and Ingredients for toppings. Training phrases can allude to parameters of the same intent. For example, the phrase "*One small pizza, please*" in line 6 refers to the parameter size using the value "*small*".

The entities keyword starts the entity declaration section (line 28). The listing declares two simple entities: PizzaSize (lines 29–33) and Ingredients (line 34). Like intents, entities should provide inputs for every language of the chatbot. The PizzaSize entity has three entries – small, medium and big – with two synonyms each. The Ingredients entity has four entries with no synonyms.

The actions section (line 35) declares the possible chatbot actions. Our chatbot has four actions: a Button response called askingForToppings (line 36), two HTTP Request post named orderPizza (line 42) and noteTopping (line 45), and a Text response named info (line 48). Button and text responses must provide an answer text for each language of the chatbot (lines 37 and 49), and in addition, button responses also define a list of button options (lines 39–40). Request actions must indicate the URL of the endpoint (lines 42 and 45), and may need to specify the data sent in the request (lines 43 and 46) as well as the type of the data (lines 44 and 47).

Finally, the section flows (line 50) defines conversation flows. The listing declares two. The first one (line 51) is a simple flow that starts with a user interaction matching the intent ToppingsInfo, followed by the chatbot answer with the information. Overall, this conversation allows users to obtain information on available pizza toppings. The second flow (lines 52–54) starts with the user intention StartOrder, and then, the chatbot asks for the pizza toppings by showing buttons with the available ingredients. At this point, the flow bifurcates, so that the user may either choose a topping (intent Toppings) or decline to add more toppings (intent EndOrder). If the user selects a topping, then, the chatbot sends the noteTopping request and goes back to ask for more toppings (label ask). Thus, the conversation contains a loop to ask and store pizza toppings, which ends when the user triggers the EndOrder intent. At that point, the chatbot sends an orderPizza request and ends the conversation.

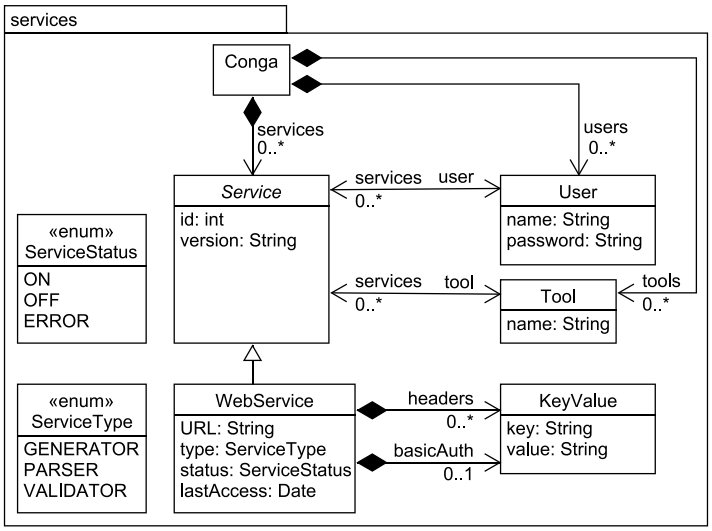

Fig. 5. Conga services meta-model.

5. Chatbot generation, parsing and validation

Next, we describe the services of Conga for the generation and parsing to/from specific chatbot tools (Section 5.1) and the validation of chatbots (Section 5.2). Then, Section 6 will explain the tool recommendation services.

5.1. Chatbot generators and parsers

Conga models are not executable, but they can be compiled into code for a particular chatbot development tool. For this purpose, our approach is extensible with chatbot compilers — generating the necessary artefacts from Conga models to specific chatbots development tools – and parsers – generating Conga models out of the artefacts of specific chatbot tools. This enables the migration of chatbots defined in a tool to another one, by using Conga as an intermediate representation.

As we discuss in Pérez-Soler et al. (2020b), there is a wide variety of tools to build and execute chatbots, each one of them with different features. Hence, to facilitate the integration of generators and parsers for arbitrary tools in Conga, we provide an extension mechanism based on web services. Fig. 5 shows the meta-model this extension mechanism relies on. The Conga platform has a list of registered Users, who can create Services for a particular chatbot Tool (identified by its name and version). WebServices are a kind of Service used to incorporate Generators, Parsers and Validators into the platform. They have an URL, a status (ON, OFF or Error), and a date of lastAccess. Technically, they may also declare a list of key–value headers and a basicAuthentication.

Generator web services receive a Conga model and return a *zip* file containing the corresponding artefacts for the target chatbot development tool. Conversely, parser web services receive a *zip* file with a tool-specific chatbot implementation, and return a Conga model.

We currently provide compilers and parsers for Rasa and Dialogflow. Table 1 shows the mapping between Dialogflow and Rasa into Conga elements. The remainder of this subsection explains this mapping in detail. In addition, for the interested reader, the Appendix provides the compilation algorithms and further details on parsing.

5.1.1. Dialogflow

Dialogflow (2024) is a low-code development platform (Di Ruscio et al., 2022) to create chatbots using a graphical web interface. Its chatbots can be exported and imported as JSON files. In this JSON-based representation, the file *Agent.json* describes global chatbot features, like its name, defined languages, or connection data to external services (the webhook). The latter include details like the URL, headers, and authentication credentials. The name and languages of the Dialogflow agent are mapped to the name and languages of the class Chatbot in Conga. The webhook becomes an HttpRequest action, mapping also the features with same name.

Entities in Dialogflow can be predefined or user-defined. The latter are described by either a regular expression, a list of literals with synonyms, or a composite entity. Each user-defined entity is exported as a JSON file containing the entity name and configuration data (if it is a regular expression or a composite entity), plus one file per defined language with the corresponding definition (literals and synonyms; regular expressions; or entities and literals in the case of composite entities). Each configuration file corresponds to a Conga Entity, and each language file to a LanguageEntity. The Entry type depends on the type defined in the configuration file. The predefined entities of Dialogflow are mapped to predefined entities in Conga.

Intents in Dialogflow have name, training phrases, responses, parameters, and an indication of whether they are fallback or enable a webhook, among other features. Intents are exported into JSON files: one configuration file per intent, with the name, responses, parameters, fallback and webhook information; and one definition file per language, with the training phrases. Each intent configuration file is mapped to a Conga Intent with Parameters. Each language definition file is mapped to a LanguageIntent and TrainingPhrases. Responses in Dialogflow correspond to Actions in Conga. We support text, image and button responses, and convert other custom responses into Empty actions.

Finally, Dialogflow controls the conversation flow via contexts. These can be input/output to intents and can store relevant conversation states. Intents without input context correspond with a UserInteraction starting a flow in Conga, and intents without output context end the flow. All intents are followed by a BotInteraction with actions corresponding to the intent responses. If an intent has enabled the webhook, the BotInteraction also has an HttpRequest action created from the webhook information.

5.1.2. Rasa

Rasa (2024) is a framework to develop chatbots using Python, markdown and YAML. The current Conga generator and parser support version 1.10.

The definition of a Rasa chatbot comprises several files. The *config.yml* file defines configuration properties, like the chatbot language or the seed NL prediction model. The *data/nlu.md* file contains training data to identify the intents correctly, as well as entity definitions. For example, the *data/nlu.md* file in Listing 2 defines an intent called StartOrder (lines 1–4). The training phrases can include parameters

Table 1
Mapping of Dialogflow and Rasa to Conga.

Dialogflow	Conga	Rasa
The *Agent.json* file contains the chatbot configuration and defined languages	Chatbot	The *config.yml* file contains the chatbot configuration and defined languages
There is a JSON file per intent, with its configuration	Intent	The file *domain.yml* contains a list with all the intents
There is a JSON file per language and intent; the language is specified within the file name	LanguageIntent	Rasa only supports one language per chatbot, which is defined in the *config.yml* file
Each language intent file contains one JSON object per training phrase	TrainingPhrase	The *nlu.md* file contains a list of training phrases per intent in markdown
The intent configuration file contains the list of parameters in a JSON array	Parameter	Parameters can be inferred from the training phrases
There is a JSON file per entity with configuration information, e.g., if it is composite or a regular expression	Entity	The *nlu.md* file may contain a list of options, synonyms of words, and regular expression for parameters
There is a JSON file per defined language and entity; the language is specified within the file name	LanguageEntity	Rasa only supports one language per chatbot, which is defined in the *config.yml* file
Each language entity file contains one JSON object per entry	Entry	–
The intent configuration file contains the intent responses (e.g., text, images, buttons)	Text, Image, ButtonAction	The *domain.yml* file contains the intent responses (e.g., text, images, buttons)
The *Agent.json* file defines the webhook information, including the URL, headers and basic authentication	HttpRequest	The HTTP request can be defined either in the *action.py* file in Python, or in the *endpoint.yml* file
The intent configuration file contains input and output contexts, which define the conversation flow	Flow	The file *data/stories.md* defines conversations flows in markdown

within brackets and followed by either the entity name in parenthesis (e.g., [Big](size)) or with curly brackets (e.g., [regular]{''entity'': ''size'', ''value'': ''medium''}). The listing also declares synonyms for literals *small* (lines 5–7), *medium* 8–10 and *big* 11–13. Each training phrase defined in file *data/nlu.md* becomes a TrainingPhrase with Parameters in Conga. The *data/nlu.md* file can also define entities, simple entities and regular expressions. Rasa also permits validating parameters directly using Python code, in which case, it is not necessary to define entities for them.

```
1  ## intent:StartOrder
2  - I want to order a [regular]{"entity": "size", "value": "medium"} pizza
3  - [Big](size) pizza
4  - One [small]{"entity": "size", "value": "small"} pizza, please
5  ## synonym:small
6  - tiny
7  - little
8  ## synonym:medium
9  - regular
10 - intermediate
11 ## synonym:big
12 - huge
13 - large
```

Listing 2: Training phrases in example *data/nlu.md* Rasa file

The file *domain.yml* lists the chatbot intents, slots and actions. Actions can be text messages, images, buttons, or custom actions defined in the Python file *actions.py*. Each action in this file is mapped to the corresponding Conga Action, and custom actions are mapped to Empty actions.

Finally, the file *data/stories.md* specifies the conversation flows. As an example, the conversation flow in Listing 3 specifies that matching the intent *ToppingsInfo* triggers the response *utter_info*. Conversation flows in Rasa are linear (without loops), which allows a direct conversion to Conga Flows.

```
1  ## story1
2  * ToppingsInfo
3      - utter_info
```

Listing 3: Flow in example *data/stories.md* Rasa file

5.2. Chatbot validation services

Conga includes validation rules that every chatbot model should satisfy. In addition, further validation web services can be added to check well-formedness constraints for specific chatbot tools (cf. Fig. 5).

Table 2 summarises the validation rules in Conga. It contains twenty general model invariants ensuring well-formedness of the chatbot models (rules G1–G20). These are further divided into *errors* and *warnings*.

Errors may be caused by equally named elements (e.g., two Actions with the same name), which is checked by G1; the use of undefined or repeated languages (e.g., an intent provides training phrases in a language that the chatbot does not support), which is checked by G2 and G3; problematic conversation flows (G4–G6); and erroneous parameter (G7) and entity (G8) definitions. Conversation flow problems may be due to having several flows starting with the same intent, which would make the chatbot unable to distinguish which flow should be followed (G4); HTTP responses with no previous HTTP request (G5); and malformed conversation loops (G6).

Warnings are classified either as syntactical issues (missing, malformed or unused elements, expected heuristics), or as best practices that ensure a proper chatbot definition. Within the first kind, warnings can signal elements (e.g., an intent) that miss phrases in some of the languages defined by the chatbot (G9); malformed regular expressions (G10); conversation loops that may not terminate (G11); and unused entities (G12), intents (G13) and parameters (G14). The rest of warnings assess the adherence to best practices for chatbot design. In particular, G15 checks that each intent defines a minimum of three training phrases since, otherwise, the intent might be imprecise (Abdellatif et al., 2022); G16 and G17 check that a training phrase is not repeated in several intents (since this might confuse the chatbot) or in the same intent; G18 checks that text parameters are not the only parameter of a phrase, as this parameter may take the whole user utterance as its value; G19 checks that the chatbot has a *fallback* intent to be used when no other intent matches the user utterance; and G20 checks that mandatory parameters define prompt phrases that the chatbot can use to ask for missing values.

In addition, Conga admits the incorporation of validation services for particular chatbot tools (currently, Dialogflow and Rasa, cf. Table 2). These services are executed before synthesising code for the chatbot tool, to ensure a proper generation. The Dialogflow validation service checks two features that the platform does not support: D1 warns that

Table 2
Validation rules in Conga (BP = Best Practice, EF = Emulated Feature, H = Heuristic, ME = Missing Element, MF = MalFormed element, NS = Not Supported, UE = Unused Element).

Id	Validation rule	Severity	Conga elem.
	General		
G1	The name of intents, actions, entities, and parameters should be unique	Error	Intent, Action, Entity, Parameter
G2	The language of the element (intent, action, entity) must be included within the chatbot languages	Error	LanguageInput
G3	There cannot be two LanguageInputs of the same language in the same element (intent, action, entity)	Error	LanguageInput
G4	Different conversation flows cannot start with the same intent	Error	Flow
G5	There should be an HttpRequest before each HttpResponse	Error	Flow
G6	Reference "back to" must point to an element in the same path, at a previous position	Error	Flow
G7	The parameters of a training phrase must be defined in the same intent	Error	Parameter
G8	All entries of the entity should be of the same type	Error	Entity
G9	There should be one LanguageInput per chatbot language	Warning (ME)	LanguageInput
G10	Regex syntax must be well-formed	Warning (MF)	RegularExpression
G11	Loops should have some terminating branch	Warning (H)	Flow
G12	Entities should be used by some parameter	Warning (UE)	Entity
G13	Intents should be used in some conversation flow	Warning (UE)	Intent
G14	Mandatory parameters should be used in some training phrase	Warning (UE)	Parameter
G15	The language intent must contain at least 3 training phrases	Warning (BP)	LanguageIntent
G16	Two training phrases should not be equal in different intents	Warning (BP)	TrainingPhrase
G17	Two training phrases should not be equal in the same intent	Warning (BP)	TrainingPhrase
G18	If a phrase has a text parameter, it should have more content than this parameter	Warning (BP)	TrainingPhrase
G19	The chatbot should have a fallback intent	Warning (BP)	Chatbot
G20	Mandatory parameters should have prompts	Warning (BP)	Parameter
	Dialogflow		
D1	The data of an HttpRequest will be ignored, Dialogflow sends its JSON format with all the information	Warning (NS)	HttpRequest
D2	Dialogflow does not support two HttpRequests in an action, only the first one is taken into account	Warning (NS)	HttpRequest
	Rasa		
R1	Rasa does not support multi-language chatbots, the generator creates one chatbot per language	Warning (EF)	Chatbot
R2	The Rasa generator unfolds the loops up to 5 repetitions	Warning (EF)	Flow
R3	Rasa does not support more than one fallback intent, only the first one is taken into account	Warning (NS)	Intent

the data of HttpRequests are ignored in Dialogflow, and D2 warns that if an action contains several HttpRequests, then only the first one will be performed. The Rasa validation service performs three validations related to limitations of the framework. First, Conga supports multi-language chatbots, but Rasa does not. Hence, R1 produces a warning explaining that the Rasa code generator will create one separate chatbot per defined language. Likewise, since Rasa does not support conversation loops, R2 warns that the generator will emulate existing loops by unrolling them five times. Finally, Rasa only supports one fallback intent, so if a Conga model defines several, R3 informs that only the first one will be used.

6. Recommending chatbot creation tools

The large number of available tools for chatbot creation makes it difficult to select the best option to build a particular chatbot. To assist in this task, Conga provides a recommender that recommends an appropriate tool to implement a chatbot, given a Conga model with its description, and the answers to a questionnaire on other aspects not captured by the model (e.g., technical, organisational or managerial requirements). The recommender has a model-based extensible architecture that enables the addition of new chatbot creation tools and the customisation of the questions and model features the recommendation builds on.

Fig. 6 shows the meta-model this 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 Options, and can be multi-response or not. In addition, Analysis requirements define an evaluator, which is a 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 requirement Options that this model fulfils. The recommendation consists of a list of tools (class ToolDescription). 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).

The recommender follows the criteria we proposed in Pérez-Soler et al. (2021), which are summarised in Table 3, but new criteria can be added if needed. The table also shows the coverage of these requirements by Dialogflow and Rasa v1.10. The analysis requirements (numbered 1 to 8 in the table) are determined by statically analysing the chatbot model to identify if the chatbot is multi-language, the targeted languages,[1] and whether it uses predefined or chatbot-specific entities, external services, phrase parameters, parameter storage, natural language processing, regular expressions, or conversation loops. Dialogflow supports all these features but cannot call to multiple external services in the same action, hence the *unavailable* value in the table. With respect to Rasa, it does not support multi-language chatbots, but the workaround that Conga implements is generating one chatbot per language, hence the value *possible* in the table. Similarly, it does not support conversation loops natively, but as explained in Section 5.2, the provided generator unrolls the loops.

Questions deal with requirements that cannot be inferred from the chatbot models, so they are explicitly asked to the developer. In particular, questions 9–15 in Table 3 concern technical aspects. They ask about 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, but others (e.g., Rasa) require an external server; the required level of support for version control, which is built-in in Dialogflow, but programming frameworks such as 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, but the free version of Rasa lacks this support); the persistence of utterances for their subsequent analysis; and the need to support speech recognition or sentiment analysis. Additionally,

[1] For brevity, Table 3 shows the number of languages supported, not the list of them.

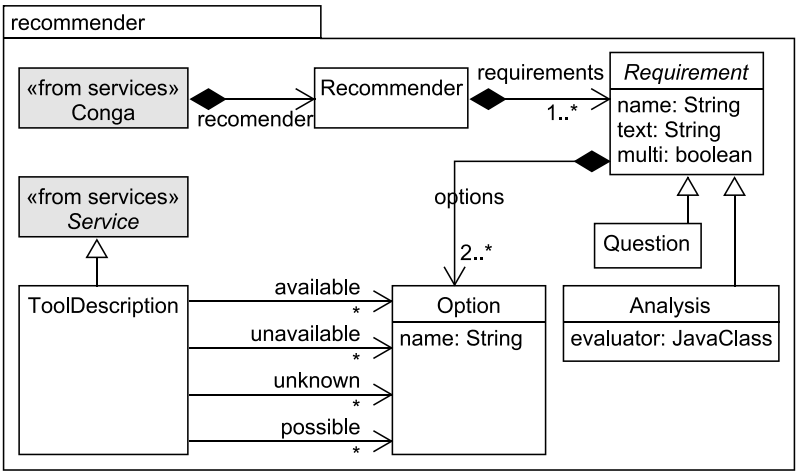

Fig. 6. Recommender meta-model.

Table 3
Requirements that the recommender currently takes into consideration.

#	Requirements	Multi-response	Options	Dialogflow	Rasa v1.10
Analyses					
1	Is the chatbot multi-language?	False	Yes No	avail. avail.	possib. avail.
2	Which are the chatbot languages?	True	–	21	all
3	Does the chatbot use new or predefined entities?	True	Predefined New None	avail. avail. avail.	avail. avail. avail.
4	Does the chatbot call to external services?	False	One Multiple None	avail. unavail. avail.	avail. avail. avail.
5	Does the chatbot use phrase parameters?	False	Yes No	avail. avail.	avail. avail.
6	Does the chatbot need persistent or volatile parameter storage?	True	Persistent Volatile None	avail. avail. avail.	avail. avail. avail.
7	Does the chatbot need NLP or pattern matching?	True	NLP Pattern matching	avail. avail.	avail. avail.
8	Does the chatbot have conversation loops?	True	Yes No	avail. avail.	possib. avail.
Questions					
9	Which social networks do you want to deploy the chatbot in?	True	–	16	8
10	Do you want to deploy the chatbot on your own host?	False	Tool host Own host	avail. unavail.	unavail. avail.
11	Do you want to use a built-in version control system?	False	Yes No	avail. avail.	avail. avail.
12	Do you require native support for chatbot analytics?	False	Yes No	avail. avail.	unavail. avail.
13	Do you require native support for utterance persistence?	False	Yes No	avail. avail.	avail. avail.
14	Do you require the chatbot to support speech recognition?	False	Yes No	avail. avail.	unavail. avail.
15	Do you require the chatbot to support sentiment analysis?	False	Yes No	avail. avail.	unavail. avail.
16	Do you require to use an open-source tool?	False	Yes No	unavail. avail.	avail. avail.
17	Which price model do you plan to use?	True	Free Pay as you go Quota Pay advanced feats.	avail. avail. unavail. unavail.	avail. unavail. unavail. avail.
18	What is the level of expertise of the development team?	False	Low High	avail. avail.	unavail. avail.

questions 16–18 tackle organisational 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, because the former requires programming.

Since some requirements may be more important than others depending on the project, they declare an importance level that the user can customise when requesting a recommendation. 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 reports the ranking of tools and how each requirement contributes to this ranking. This way, the recommender considers multiple objectives (as many as evaluated requirements, cf. Table 3) and the user configures their relevance in his/her context. Moreover, the generated ranking of recommendations is specific for each chatbot model (as it is based on the analyses conducted on it) and context of use (as the questions are answered within that context). For practical purposes, the recommender caches the calculated ranking for each chatbot model. This cache is invalidated whenever the chatbot model or the recommendation configuration change (e.g., after adding new Requirements or ToolDescriptions).

Algorithm 1 captures the explained recommendation procedure. Its input is a mapping *importance* classifying the requirements in Table 3 as irrelevant, relevant, double relevant or critical; and a mapping *answer* containing the selected options for those requirements (e.g., Yes, No). Since some requirements admit non-exclusive answers (multi-response in Table 3), *answer* maps a set of options, and not just one. Mapping *importance* is set by the user depending on the project context. Mapping *answer* contains entries automatically derived by the analysis of the chatbot model (items 1–8 in Table 3), and entries set by the user (items 9–18 in Table 3).

Algorithm 1 Generate Platform Recommendation for a Chatbot Model

```
Input: importance : Requirement → {irrelevant, relevant, double relevant, critical}
Input: answer : Requirement → P(Option)
Output: rankedTools : Tool → ℝ
1: requirements ← load all requirements
2: tools ← load all tool descriptions
3: scores ← empty map from tool to score
4: for all tool ∈ tools do
5:     toolScore ← toolScore(tool, requirements, importance, answer)
6:     scores[tool] ← toolScore if toolScore ≠ none
7: end for
8: rankedTools ← sort tools by descending scores
9: return rankedTools
10:
11: function toolScore(tool, requirements, importance, answer)
12:     score ← 0
13:     for all req ∈ requirements do
14:         if importance[req] ≠ irrelevant then                ▷ skip irrelevant requirements
15:             reqScore ← 0
16:             for all option ∈ answer[req] do
17:                 status ← tool's status for option           ▷ avail., unavail., unknown, possible
18:                 return none if importance[req] = critical and status ≠ available   ▷ tool skipped
19:                 f ← 1 if importance[req] is relevant or critical, 2 if double relevant
20:                 c ← 1.0 if status = available, 0.75 if possible, 0.5 if unknown, 0 otherwise
21:                 reqScore ← reqScore + (f × c)
22:             end for
23:             score ← score + reqScore/size(answer[req])
24:         end if
25:     end for
26:     return score/size(requirements)
27: end function
```

The output of Algorithm 1 is a set of tools ranked by their score (lines 8–9). The score of a tool is calculated by the auxiliary function toolScore (lines 11–27). This function iterates on the set of requirements that are not irrelevant (line 14), assessing the selected options for the requirement (lines 16–22). If the requirement is critical but the selected option is not available in the tool, the tool is discarded (line 18). Otherwise, a partial score is calculated multiplying the importance f of the requirement (line 19) and the status c of the option (line 20). Finally, the score of the tool is the sum of partial scores obtained for each requirement and option, averaged over the number of requirements and options (lines 23 and 26).

In terms of extensibility, the recommender can be extended in three ways, depicted in Fig. 7. First, its model-based architecture allows adding new automatic analyses that contribute to the calculation of the recommendation score. This is done by adding an Analysis object to the recommender model, defining a Java evaluator method with its output options, and specifying the support by the registered tools. For instance, to add an analysis checking whether the conversation flow of a chatbot has bifurcations, the possible options are *yes* and *no*, and bifurcations are *available* in Dialogflow and Rasa (cf. Fig. 7(a)). Second, it is possible to add new questions to the user, together with their answer options and their support by the registered tools. For example, one may add the question *Do you require generative AI in chatbot outputs?* with answer options *yes* and *no*, and being this feature *unavailable* in Dialogflow and Rasa (cf. Fig. 7(b)). Third, new ToolDescriptions (e.g., Watson) can be added to the recommender. This entails informing their support for the analyses and questions in the recommender, so that the recommendations remain comprehensive as new tools are introduced (cf. Fig. 7(c)). This architecture avoids the need to hard-code any analysis, question or tool, but all the knowledge is added via an extensible model.

Overall, incorporating a new chatbot creation tool (e.g., Watson) into our platform requires: (i) providing a code generator from Conga to the tool, and optionally, a validation service; (ii) providing a parser if reverse engineering is required; and (iii) informing the tool options for every requirement in the recommender (cf. Table 3). Our platform 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 emulated. For instance, Rasa does not support multi-language chatbots, but this can be emulated by generating one chatbot per language.

7. Architecture and tool support

Next, we describe the architecture of Conga (Section 7.1), the tool support in the form of a web platform (Section 7.2), and how to extend Conga to support new chatbot platforms (Section 7.3).

The code of Conga is released as open source at https://saraperezsoler.github.io/CONGA/. The tool is deployed at http://dimo1.ii.uam.es/CONGA/ (it has a test user *test1* with password *testOne*). A video showcasing the tool is available at https://www.youtube.com/watch?v=3sw1FDdZ7XY.

7.1. Architecture

Conga is a web application targeted to chatbot developers. Fig. 8 shows its architecture. The front-end includes a project manager that allows creating and deleting projects; a service manager for registering external services for chatbot validation, code generation and parsing; and a recommender manager that enables the creation of a tool description for any registered code generator. In addition, the front-end offers an editor of Conga models, a graphical renderer of conversation flow models, a questionnaire for the tool recommender, a visualiser of tool recommendation reports, and importers, exporters and validators for Dialogflow and Rasa.

The back-end handles the requests of the front-end. The managers store the information in a storage model, explained below. The chatbot

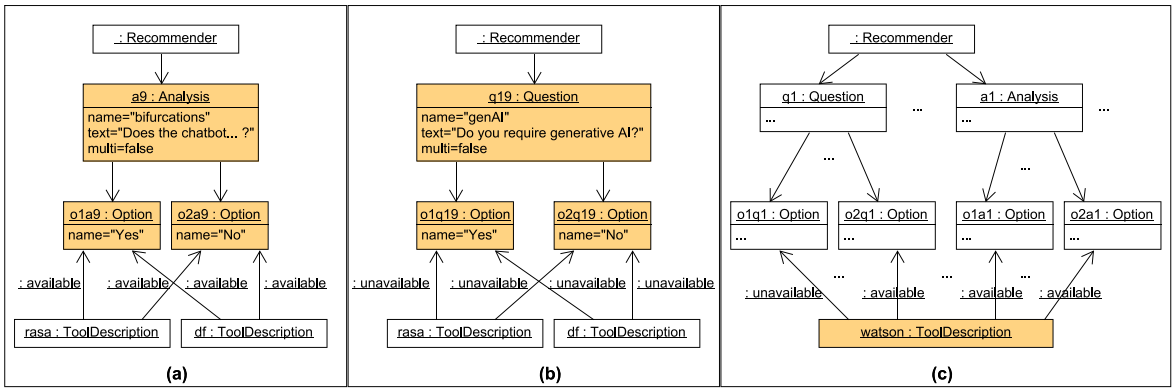

Fig. 7. Extending the recommender (added elements are coloured in orange). (a) Adding a new analysis. (b) Adding a new question. (c) Adding a new tool description.

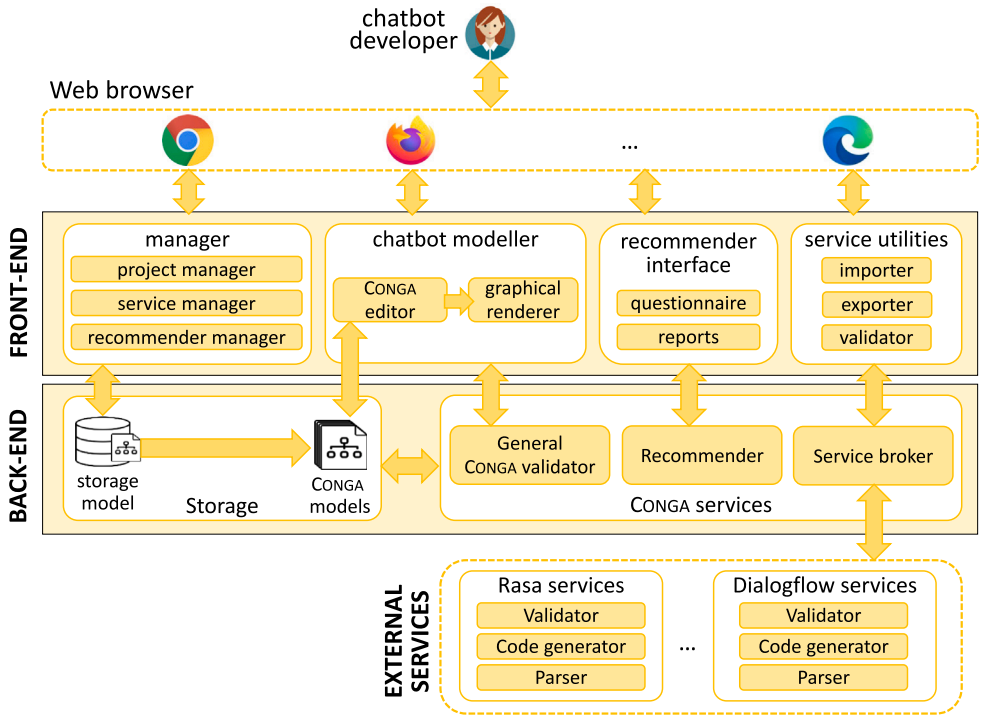

Fig. 8. CONGA's architecture.

models created in the editor are stored as CONGA files. The back-end also contains services for general chatbot validation, recommendation computation, and a service broker that handles the calls to the registered validation, code generation and parsing services of particular tools. Code generators and validators receive a CONGA model in XMI format,[2] and return a *zip* file containing the tool-specific implementation or a JSON file with the errors, respectively. Conversely, parsers receive a *zip* file with the tool-specific implementation of a chatbot, and return a CONGA model in XMI format.

The storage model of CONGA conforms to the meta-model of Fig. 9. Each Chatbot definition is stored in a Project. Projects have an owner User, and store the answers of the user to the recommender questionnaire (class RecomAnswer) for the defined chatbot. Each answer (class QuestionAnswer) has a reference to the answered question, to the selected option for that question, and stores its importance level for the recommendation (*irrelevant*, *relevant*, *double relevant*, *critical*, or *critical all* to mark all selected options in multi-response questions as critical). The importance level is not decided a priori, but by the developer who seeks the recommendation. This is so as the relevance of a chatbot feature may depend on the specific project. In addition to the answers, the project stores the ranking of tools computed by the recommender, including the score for each ToolDescription.

7.2. Tool support

Fig. 10 shows the interface of CONGA. The header (label 1) includes a link to the documentation (hosted in GitHub), menus for the project and service managers, the name of the logged user, and a button to sign out. The tool bar (label 2) has buttons for saving the chatbot model being edited, creating a new project, formatting the displayed file, validating the chatbot with one of the registered tool-specific validators, selecting a development tool to generate code for, filling in the tool recommender questionnaire, and displaying the recommendation

[2] XMI is the default persistence format of the Eclipse Modelling Framework (Steinberg et al., 2008), which our platform uses to describe the CONGA meta-model, and the chatbot models.

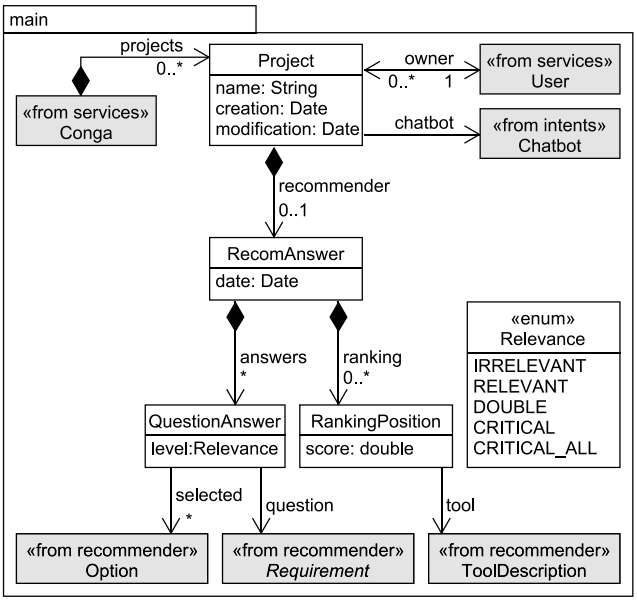

Fig. 9. Conga storage meta-model.

results. New projects can be created empty, or be populated with a model parsed from an existing chatbot implementation.

In the figure, the editor (label 3) contains the example pizzeria chatbot from Listing 1. The editor is displayed in a text area with line numbers, and features syntax highlighting, content assistance and error reporting. In the figure, the editor reports some warnings. The warning/error messages include their source (general error, or tool-specific error) in brackets. For instance, the warnings in the figure are reported by the Dialogflow validator. They warn that Dialogflow only allows one HTTP action (rule D1 of Table 2), and its data will be ignored (rule D2 of Table 2). Technically, the editor is implemented in Xtext (2024), and uses its web deployment options for the CodeMirror JavaScript library (CodeMirror, 2024). In addition, a *problems console* (label 4) groups all warnings/errors by their source, indicating their line and message. The messages use the following colour code: green for a successful validation, yellow for warnings, and red for errors.

The diagram to the right (label 5) depicts the conversation flow of the chatbot model graphically. It represents the user interactions as transitions, and the chatbot interactions as states with the actions that the chatbot performs inside. This diagram is created using PlantUML (2024), and gets updated upon saving the chatbot.

Fig. 11 shows the tool recommender assistant. The left part presents an excerpt of the questionnaire that developers must answer to obtain tool recommendations. The questionnaire is created on-the-fly, dynamically based on the modelled requirements (cf. Fig. 6). This means that new requirements or tools can be added at any point, and the recommendation algorithm will consider them. Each question has a list of options and a selector of relevance. The specific relevance level selected does not influence in the interaction with the questionnaire, but instead the recommendation algorithm uses it to calculate the ranking of tools. The right side displays the produced ranking of tools ordered by decreasing scores. Clicking on the button to the right of a tool invokes the corresponding code generator and the resulting artefacts can be downloaded.

Recommendations are not black boxes, but developers can inspect their rationale in an explainability table, shown in Fig. 12. This table reports, for each available tool, its percentage of support of every requirement, marking in grey the non-relevant ones. This facilitates the feature-based comparison of tools. There is another view, shown in Fig. 13, with detailed information per available tool, namely, the result of the analyses for the tool, the answers to each question, the relevance assigned to the analyses and questions, and their contribution to the recommendation score.

Fig. 14 shows the service manager, which lists the registered services. For each service, it shows its type (*recommender*, *generator*, *validator*, or *converter* to indicate parsers), the tool name, the service version, the location,[3] the last access date to the location, the status (*ON*, *OFF* or *error* in case the service does not work correctly), and buttons to (de-)activate, edit and remove the service.

7.3. Extensibility

Conga's modular architecture permits keeping up to date with the evolving ecosystem of chatbot technologies. Adding full support for a new chatbot technology entails building a parser, a generator, a validator (if special rules are needed), and providing an answer to the recommender questions for the technology, in the form of a model. This task should be performed by engineers with knowledge of the chatbot technology and Conga's model.

For reference, Table 4 shows size metrics of these services for Dialogflow and Rasa. Parsers are the largest services as they need to handle the specific serialisation format employed by the chatbot technology (JSON for Dialogflow; YAML, markdown, and Python for Rasa). Our strategy was to create domain Java classes for the technology-specific concepts (cf. Appendices A.3 and A.4), and organise the translation into Conga as a generic model-to-model transformation. We built the generators (from Conga to the technologies' serialisation format) using Xtend,[4] a template language to emit text from models. Finally, we implemented the validators in Java, taking advantage from the abstract class and infrastructure that Conga provides to facilitate their construction.

It must be noted that each chatbot technology has its own features (Pérez-Soler et al., 2021), so there is the risk that some of them cannot be directly mapped to Conga constructs for being too specific or expressive. A workaround would be needed to handle this situation. For example, let us imagine that Conga lacks multi-language support (though this is not the case). Then, the parser from Dialogflow (which admits multi-language chatbots) into Conga should create a different Conga chatbot per supported language, replicating the same chatbot structure but using training phrases in different languages. Conversely, if a feature available in Conga (e.g., loops) is unavailable in a chatbot technology (e.g., Rasa), the generator may introduce a workaround if possible (e.g., unrolling the loop several times).

In any case, the recommender keeps the list of potential features in chatbot technologies, represented as individual Analysis objects for each feature. Then, each technology declares whether it supports each feature natively, through a workaround, or lacks support entirely. If a Conga chatbot uses a feature unsupported by a tool, the tool will not be recommended, preventing erroneous code generation. The set of tool features is not closed, but the recommender can be extended with new Analysis objects at any time.

While we provide full parsing and generation support for Rasa and Dialogflow, we have used the extensibility mechanism of the recommender to model 11 additional chatbot tools: Watson, Lex, Microsoft Bot Framework, SmartLoop, Xenioo, LUIS, FlowXO, Landbot.io, ChatterBot, Chatfuel, Botkit, and Pandorabots. The recommender model with these tools is available at: https://github.com/SaraPerezSoler/CONGA/blob/master/RecomenderModel.xmi.

[3] The *recommender* service has no location because it is built-in within the platform.

[4] https://eclipse.dev/Xtext/xtend/.

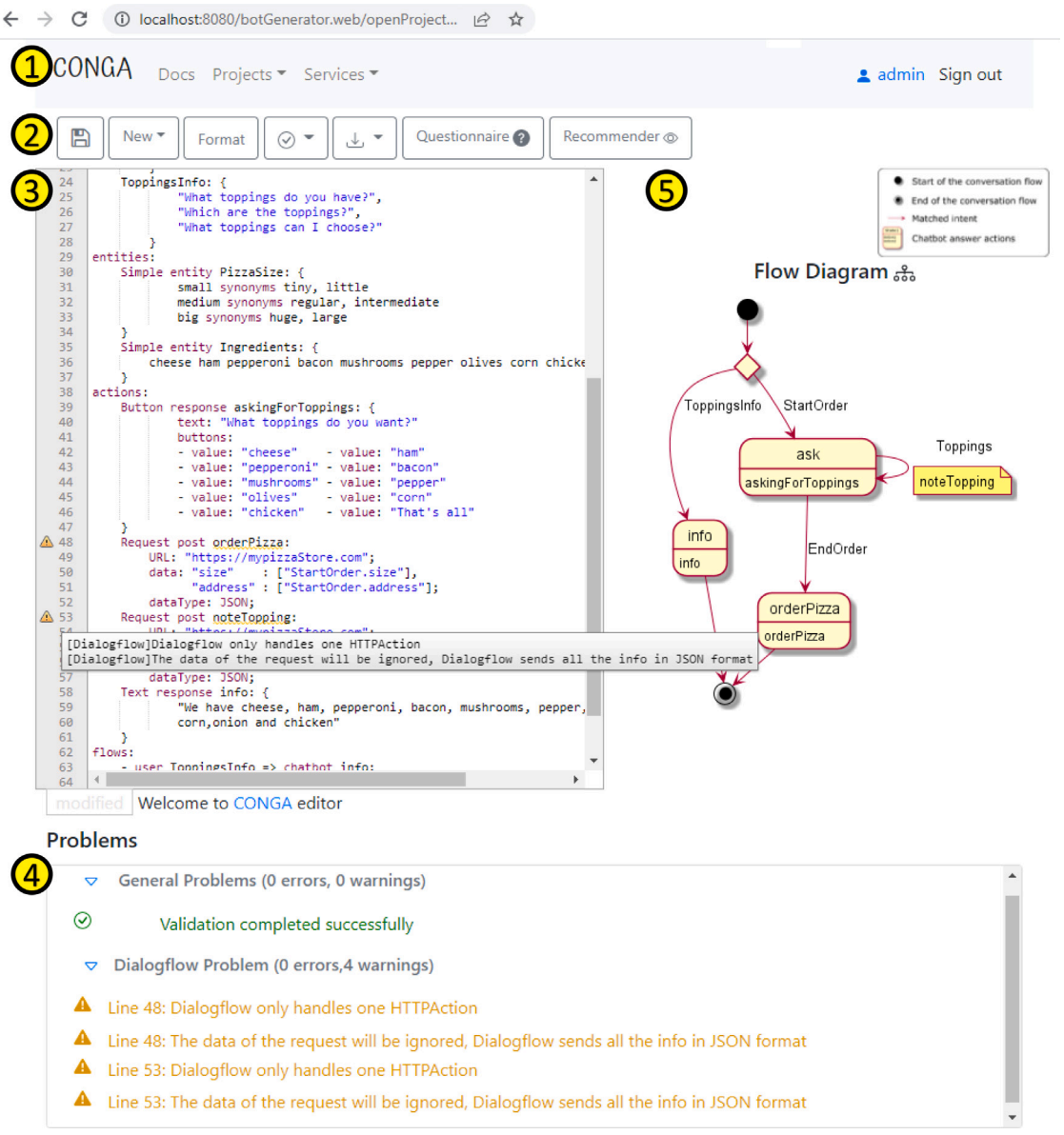

Fig. 10. CONGA's main interface.

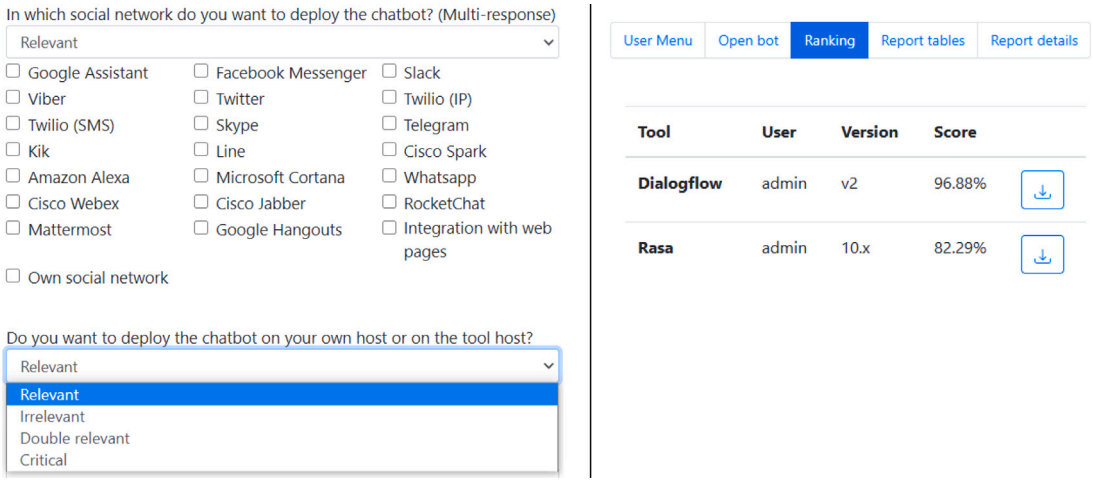

Fig. 11. CONGA recommender support. Requirements questionnaire (left side). Resulting ranking of tools (right side).

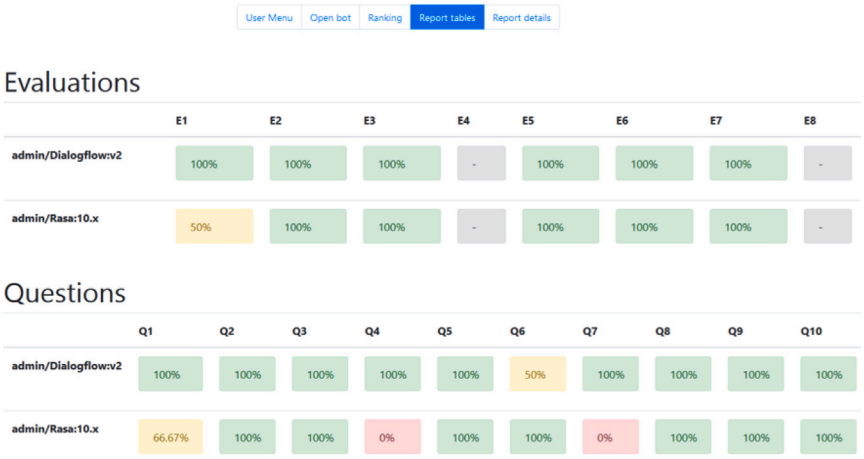

Fig. 12. Recommendation explainability table.

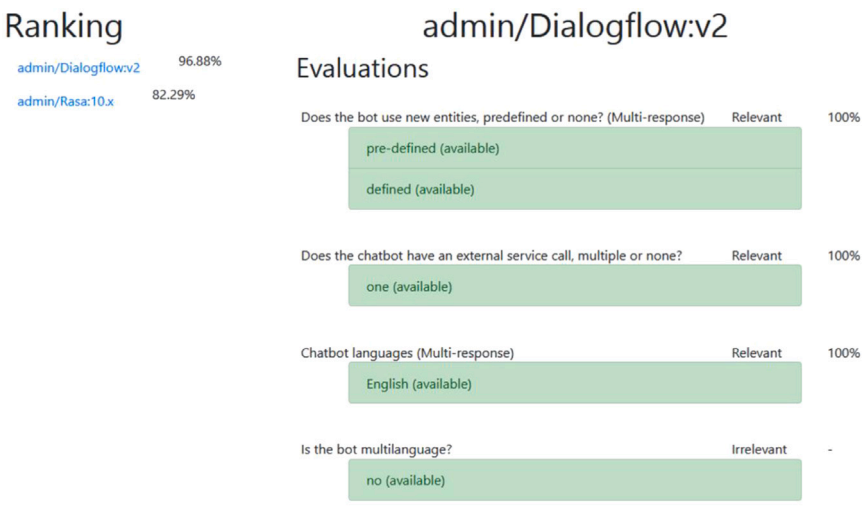

Fig. 13. Recommendation score details.

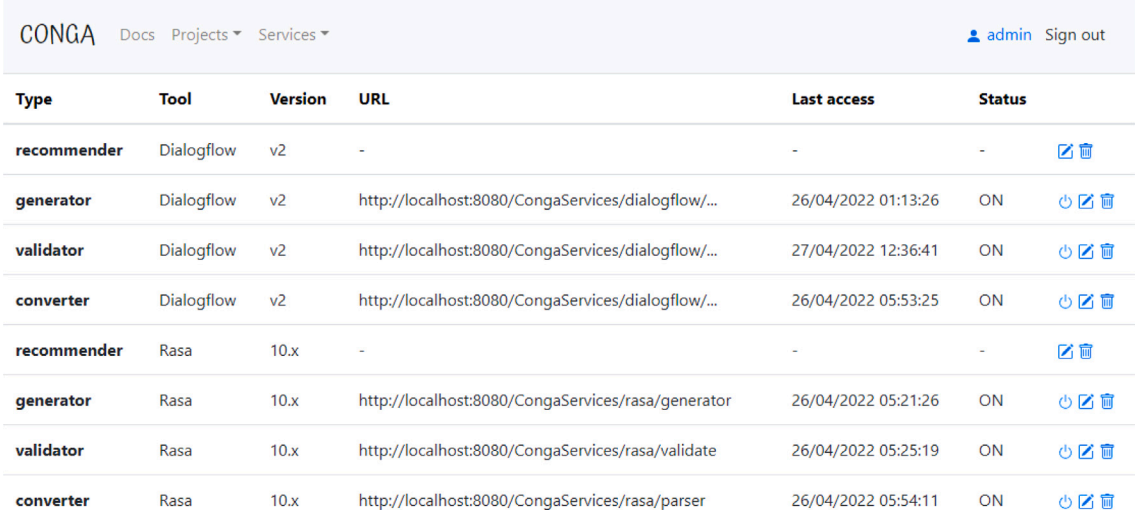

Fig. 14. CONGA service manager.

Table 4
Size of the Dialogflow and Rasa services.

	Dialogflow	Rasa
Parser (Java LoC)	1485 (16 classes)	2371 (32 classes)
Generator (Xtend LoC)	667 LoC	820 LoC
Validator (Java LoC)	113 LoC	92 LoC
Recommender (model elements)	72 av, 13 unav (85 opt)	61 av, 7 unav, 15 unk, 2 pos (85 opt)

Table 5
Summary of the chatbots dataset.

Technology	Source	#Chatbots
Dialogflow	Dialogflow	17
	GitHub	116
	Kommunicate	7
Rasa	Rasa	3
	GitHub	148
	Total	291

8. Evaluation

Next, we evaluate our approach to assess the quality assurance and migration scenarios. The evaluation aims at answering the following research questions (RQs):

RQ1 Can Conga help detecting quality issues in existing chatbots?
RQ2 Is Conga expressive enough to capture the details of Dialogflow and Rasa chatbots?
RQ3 To what extent does Conga automate the migration between Rasa and Dialogflow?

For this purpose, we have performed three experiments based on a set of 151 Rasa and 140 Dialogflow open-source chatbots developed by third parties. The chatbots used in the experiments and the raw data are available at https://github.com/Conga-dsl/ValidationSetup.

Next, Section 8.1 presents the setup of the experiments, Sections 8.2–8.4 answer the RQs, Section 8.5 discusses the implications of the results, and Section 8.6 analyses threats to validity.

8.1. Experiment setup

To perform the evaluation, we collected 291 chatbots (140 of Dialogflow, 151 of Rasa version 1.10) developed by third parties. We used four sources: GitHub to collect open-source chatbots; the Dialogflow platform, as it includes pre-defined template agents; the Kommunicate website,[5] which includes examples of Dialogflow agents; and the Rasa framework website, to obtain Rasa chatbot examples. We checked for duplicates, making sure there were none. Table 5 summarises the content of the dataset, which is available at https://github.com/Conga-dsl/ValidationSetup.

To obtain a better picture of the dataset, Table 6 summarises the chatbots' features, displaying the average, median, maximum and minimum number of defined languages, intents, entities, actions, conversation flows, specific action kinds (e.g., HTTP requests), and conversation loops. The last three columns show the percentage of chatbots with zero, exactly one, and more than one of these elements.

In summary, most Dialogflow chatbots in the dataset support one language (normally English), but still 8,57% of them support several ones, peaking at 12. Instead, all Rasa chatbots define just one language because Rasa does not support multi-language chatbots. Rasa v1.10 does not support conversation loops, while 11,43% of the Dialogflow chatbots define loops. The intent and flow metrics are similar in Dialogflow and Rasa. Rasa chatbots declaratively define significantly fewer entities than Dialogflow, as most of them are defined with Python code, as explained in Section 5.1. The same happens with HTTP requests in Rasa: they are captured as Empty actions in Conga, because they are specified in Python code. In general, the Empty actions for Rasa represent actions specified in Python that Conga cannot parse. These can be either Rasa actions or Rasa forms. Although Dialogflow chatbots may have Empty actions as well (due to specific actions of the deployment channel), the dataset contains none. All Dialogflow chatbots contain HTTP requests, however, some of them are undefined, with an empty URL.

Table 6
Aggregate report of the chatbots' features.

	Avg	Med	Max	Min	% = 0	% = 1	% > 1
Dialogflow							
Languages	1,16	1	12	1	0,00%	91,43%	8,57%
Intents	12,51	7	94	1	0,00%	0,71%	99,29%
Entities	2,81	1	37	0	45,00%	12,14%	42,86%
Actions	10,73	6	122	2	0,00%	0,00%	100%
Flows	10,40	5	89	1	0,00%	1,43%	98,57%
Buttons	0,03	0	3	0	98,57%	0,71%	0,71%
Texts	8,66	4	119	0	16,43%	8,57%	75,00%
Images	0,04	0	3	0	97,86%	0,71%	1,43%
Empty	0,00	0	0	0	100,00%	0,00%	0,00%
Requests	1,00	1	1	1	0,00%	100%	0,00%
Loops	1,46	0	66	0	88,57%	1,43%	10,00%
Rasa							
Languages	1,00	1	1	1	0,00%	100%	0,00%
Intents	16,77	9	150	2	0,00%	0,00%	100%
Entities	0,39	0	9	0	82,12%	7,95%	9,93%
Actions	20,81	14	204	4	0,00%	0,00%	100%
Flows	8,38	3	103	1	0,00%	14,57%	85,43%
Buttons	0,97	0	14	0	65,56%	5,96%	28,48%
Texts	13,99	8	110	1	0,00%	1,99%	98,01%
Images	0,39	0	5	0	68,21%	27,81%	3,97%
Empty	5,44	2	203	0	15,23%	21,19%	63,58%
Requests	0,00	0	0	0	100,00%	0,00%	0,00%
Loops	0,00	0	0	0	100,00%	0,00%	0,00%

8.2. RQ1: Can Conga help detecting quality issues in existing chatbots?

To answer this question, we converted the chatbots into Conga models and executed the general validators. Table 7 reports the results, which show issues for the chatbots in both technologies. Most are warnings, but there are three errors in one Dialogflow chatbot. This is the chatbot *HR-Bot*, which contains three elements with text in languages that are not among the languages the chatbot declares.

Overall, the number of issues in Rasa and Dialogflow chatbots are similar, with Dialogflow having 99 more problems in total. In Rasa, warning G13 (*intents should be used in some conversation flow*) has the highest number of occurrences (443), and warning G19 (*the chatbot should have a fallback intent*) is the issue that more chatbots have (>80%). Defining intents and not using them in a conversation flow (G13) is an error more common in Rasa, probably because intents and conversation flows are defined in different files in Rasa, but in Dialogflow they are defined together using the intent context. The lack of a fallback intent (G19) causes the chatbot not to respond when the

[5] https://docs.kommunicate.io/docs/bot-samples.

Table 7
Issues found by Conga's validator.

Rule	Dialogflow			Rasa		
	#chatbots	%chatbots	#Total	#chatbots	%chatbots	#Total
Errors						
G1	0	0,00%	0	0	0,00%	0
G2	1	0,71%	3	0	0,00%	0
G3	0	0,00%	0	0	0,00%	0
G4	0	0,00%	0	0	0,00%	0
G5	0	0,00%	0	0	0,00%	0
G6	0	0,00%	0	0	0,00%	0
G7	0	0,00%	0	0	0,00%	0
G8	0	0,00%	0	0	0,00%	0
Warnings						
G9	71	50,71%	756	49	32,45%	260
G10	0	0,00%	0	5	3,31%	5
G11	4	2,86%	10	0	0,00%	0
G12	32	22,86%	58	16	10,60%	34
G13	8	5,71%	45	84	55,63%	443
G14	13	9,29%	45	0	0,00%	0
G15	66	47,14%	250	40	26,49%	103
G16	8	5,71%	110	17	11,26%	160
G17	35	25,00%	68	65	43,05%	226
G18	30	21,43%	43	75	49,67%	108
G19	7	5,00%	7	122	80,79%	122
G20	22	15,71%	165	0	0,00%	0
Total						
	–	–	1560	–	–	1461

user utterance does not match any existing intent, frustrating the user. G19 hardly occurs in Dialogflow because this technology sets a fallback intent by default, whereas in Rasa, the fallback must be explicitly defined in the configuration file, which may be less intuitive.

In Dialogflow, the most common issue is G9 (*there should be one language input per chatbot language*), followed by G15 (*the language intent must contain at least three training phrases*). Warning G9 pinpoints to chatbot elements lacking text in some of the languages that the chatbot declares. This is troublesome as the chatbot will not be able to respond properly in those languages. Warning G15 denotes an intent with a low (<3) number of training phrases, so the chatbot might have trouble to recognise it.

Some issues only appeared in Rasa chatbots or in Dialogflow chatbots, but not in both. Specifically, G10 (*regex syntax must be well-formed*) occurred only in Rasa (5 chatbots); while G11 (*loops should have some terminating branch*), G14 (*mandatory parameters should be used in some training phrase*) and G20 (*mandatory parameters should have prompts*) were only present in Dialogflow (4, 13 and 22 chatbots, respectively). Issue G11 does not occur in Rasa because Rasa does not support conversation loops. Issue G14 warns that an intent has no training phrases exemplifying some of the intent's parameters, so the chatbot may not learn how to recognise the parameters properly. Issue G20 detects a lack of prompts to explicitly ask the user for a non-informed parameter value.

The graphic in Fig. 15 shows the percentage of chatbots per issue type. Altogether, 10,7% of Dialogflow chatbots and 2% of Rasa chatbots are issue-free (15 and 3 respectively, an overall 6,2%). The maximum number of issue types in a chatbot is 7 in both cases (2,9% of the Dialogflow chatbots and 1% of the Rasa chatbots). Most Dialogflow chatbots have 1 or 2 issue types, and most Rasa chatbots have issues of 2 to 4 types.

Fig. 16 displays the percentage of chatbots per total number of issues. Most chatbots have between 0 and 9 issues in both cases, and the percentages decrease as the number of issues increases. The maximum number of issues in a Dialogflow chatbot is 464, and in Rasa is 292.

The complete results of this experiment are available at https://github.com/Conga-dsl/ValidationSetup.

Answering RQ1. We can answer RQ1 positively because, as Table 7 shows, the validation service of Conga found issues in most chatbots of the dataset (in 93,8%). Moreover, every kind of warning (G9–G20) was found in at least one chatbot. Only 10,7% of the analysed Dialogflow chatbots, and 2% of the Rasa chatbots, are free of problems. This suggests the need of quality assurance mechanisms – like those supported by Conga – for chatbot designs.

8.3. RQ2: Is Conga expressive enough to capture the details of Dialogflow and Rasa chatbots?

To answer this question, we follow the methodology in Fig. 17. First, we convert Rasa and Dialogflow chatbots of the dataset into Conga models. Then, to assess if Conga is able to capture all the relevant details of the chatbots, we map them back to their original platform (step 2 in the figure). Finally, we compare the original and generated chatbots. However, a syntactical comparison is not enough to assess information preservation, as it would consider irrelevant serialisation details (e.g., the order of the training phrases in the file), or normalisation factors applied when parsing the chatbot into Conga. Instead, we apply differential testing (McKeeman, 1998) – a common technique for testing compilers (Chen et al., 2021) – to perform a *behavioural comparison* which checks that no information is lost and behaviour is preserved. Differential testing is generally used to test two or more versions of a system against a test suite (which may be generated automatically, or manually). Then, an error is detected if the result of running a test is different in the considered versions. In our context, we create tests for the original chatbot that accurately describe its behaviour, and execute them on the generated one (label 3 in the figure). If the tests fail on the generated chatbot, then the behaviour is not preserved, meaning that Conga did not capture properly some detail of the original chatbot.

Algorithm 2 describes the differential testing procedure in detail. It receives two chatbots c_1 and c_2, and a test suite S passing on c_1. The test suite S contains test conversation scenarios (i.e., sequences of user inputs and expected chatbot outputs) that model the behaviour of c_1. The output of the procedure is the subset of test scenarios from S that fail on c_2, indicating divergent behaviour from c_1. If such a subset is empty, then the behaviours of c_1 and c_2 are indistinguishable using S. To calculate this subset, the procedure iterates on every *test* in S, and if the execution of *test* on c_2 yields a failure, then *test* is added to the subset *result* of failing tests.

Algorithm 2 Differential Testing of Two Chatbots

Input: c_1, c_2: Chatbots
Input: S: Set of test cases passing on chatbot c_1
Output: *result*: Set of test cases from S failing on chatbot c_2
1: $result \leftarrow \emptyset$
2: **for all** $test \in S$ **do**
3: $response \leftarrow$ Execute *test* on c_2
4: **if** $response =$ failure **then**
5: Add *test* to *result*
6: **end if**
7: **end for**
8: **return** *result*

Since behavioural comparison is computationally expensive, we selected ten Dialogflow chatbots and ten Rasa chatbots from the dataset presented in Section 8.1. Table 8 shows the selected chatbots and their features.

The selection criterion was the coverage of all aspects of Conga. First, attending to the number of intents and actions, we selected large (e.g., *HR-Bot*, *Data-Mining-Chatbot*), medium (e.g., *FAQ*, *santoshkumar30*) and small chatbots (e.g., *city-streets-trivia*, *RasaProject_Docker*). In addition, some chatbots use buttons (e.g., *vardhaman-freshers*), images (e.g., *diagrams2ai*), and Empty actions (e.g., *RasaProject_Docker*).

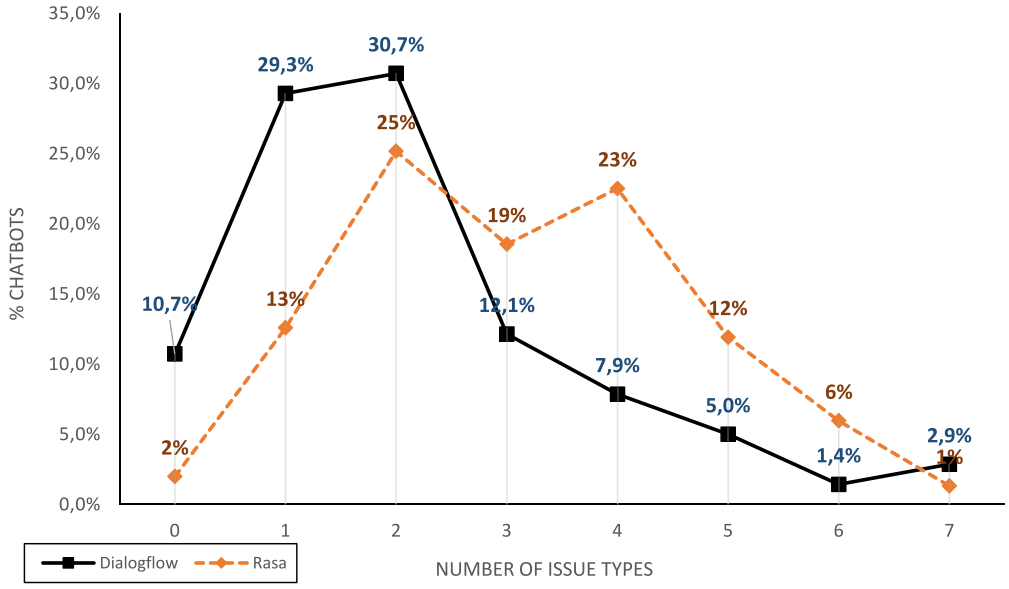

Fig. 15. Percentage of chatbots per number of issue types.

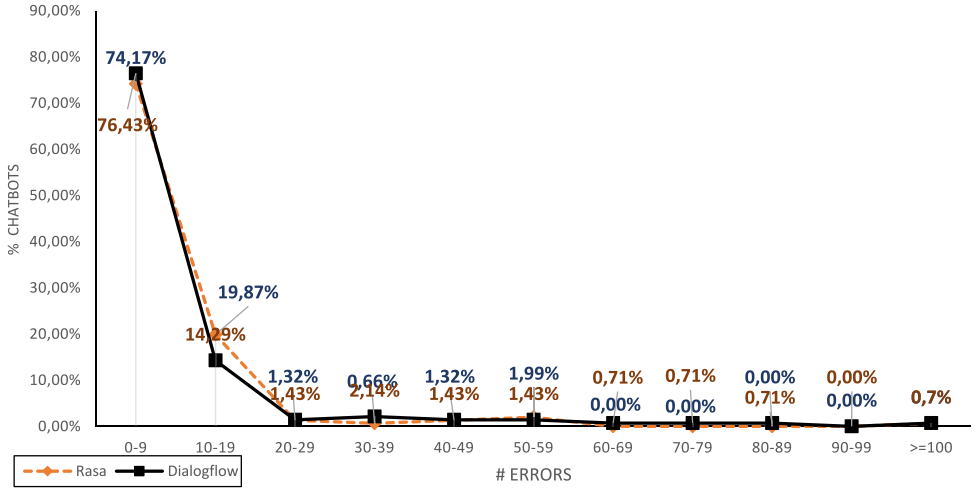

Fig. 16. Percentage of chatbots per number of issues.

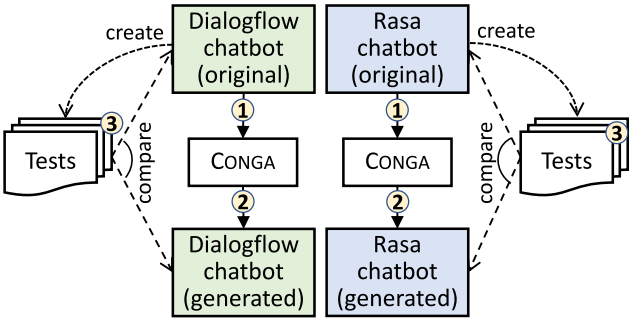

Fig. 17. Research methodology to answer RQ2.

The latter correspond to Rasa actions expressed in Python code, which CONGA is able to identify but not to interpret in the general case (e.g., Python code modifying the conversation flow dynamically). With respect to the conversation flow, we selected chatbots with less flows than intents, which leads to long conversation paths (e.g., *vardhaman-freshers*), and chatbots with similar number of flows and intents, which yields short conversation paths (e.g., *woman*). In addition, some Dialogflow chatbots have conversation loops (*city-streets-trivia* and *G7-CinemaTicketBot*) or several fallback intents (*city-streets-trivia*), two features that Rasa v1.10 does not support. Finally, among the chatbots covering these aspects, we chose those with the least number of issues. From the selected chatbots, the ones with the highest number of issues are *HR-Bot* for Dialogflow (26 issues) and *Spaceonova_Chatbot* for Rasa (31 issues).

We converted the twenty selected chatbots into CONGA, and these again to the source platform. For three Rasa chatbots (*diagrams2ai*, *Email-WhatsApp-Integration-Chatbot* and *vardhaman-freshers*), we had to

Table 8
Features of the selected chatbots for RQ2.

Name	#Intent	#Entity	#Action	#Button	#Img	#Empty	#Flow	#Loop	#Fallback	#Issue
Dialogflow										
city-streets-trivia	4	2	7	0	0	0	3	1	2	0
Coffee-Shop	15	8	16	0	0	0	6	0	1	0
HR-Bot	20	15	30	3	1	0	12	0	1	26
G7-CinemaTicketBot	11	6	11	0	0	0	3	4	1	1
iLearn_Dialogflow	6	0	7	0	0	0	4	0	1	0
fulfilment-regex	5	0	7	0	0	0	3	0	1	0
FAQ	10	4	10	0	0	0	5	0	1	2
woman	10	0	11	0	0	0	10	0	1	4
Education_Chatbot	16	0	16	0	0	0	16	0	1	13
pizzabot	5	6	5	0	0	0	5	0	1	2
Rasa										
diagrams2ai	7	0	6	0	1	0	3	0	1	0
Email-WhatsApp-Integration-Chatbot	9	0	11	0	2	3	4	0	1	0
RasaProject_Docker	6	0	7	0	0	2	4	0	1	0
256644	11	0	12	0	1	0	10	0	0	11
yassinelamarti	9	0	8	0	1	0	4	0	0	1
Data-Mining-Chatbot	42	0	43	0	1	0	38	0	0	1
nep-chatbot	15	0	15	0	0	0	15	0	0	2
santoshkumar30	12	0	18	0	1	1	8	0	0	3
vardhaman-freshers	35	0	30	2	0	0	4	0	1	7
Spaceonova_Chatbot	16	0	15	0	1	0	14	0	0	31

Table 9
Results of the evaluation of RQ2 and RQ3.

Name	Tests RQ2		Tests RQ3	
	#Passed	#Failed	#Passed	#Failed
Original chatbot in Dialogflow				
city-streets-trivia	3 219	0	99	3120
Coffee-Shop	22 884	0	22 884	0
HR-Bot	1 762	0	1 762	0
G7-CinemaTicketBot	4 630	0	4 630	0
iLearn_Dialogflow	147	0	147	0
fulfilment-regex	1 728	0	1 728	0
FAQ	24 648	0	24 648	0
woman	37	0	37	0
Education_Chatbot	36	0	36	0
pizzabot	2 290	0	2 290	0
Original chatbot in Rasa				
diagrams2ai	2 212	0	2 212	0
Email-WhatsApp-Integration-Chatbot	1 319	0	1 319	0
RasaProject_Docker	40	252	40	252
256644	107	0	107	0
yassinelamarti	725	0	725	0
Data-Mining-Chatbot	1 801	0	1 801	0
nep-chatbot	79	0	79	0
santoshkumar30	812	84	812	84
vardhaman-freshers	6 214	0	6 214	0
Spaceonova_Chatbot	131	0	131	0

manually adjust the fallback intent threshold that the code generator sets by default. This was the only adjustment performed in the generated code.

Next, we created the tests using Botium (2024), a black box testing tool that runs test scripts on the chatbot under test, and asserts whether the obtained results are the expected ones. Tests in Botium consist of scenarios made of sequences of interactions between the user and the chatbot. We created the tests manually, based on the behaviour of the original chatbot, and ensuring a maximal coverage of the conversation flows and training phrases to capture accurately the behaviour of the original chatbot (Cañizares et al., 2024a). The tests cover all possible conversation flows, for every combination of training phrases and parameter values. For chatbots *Coffee-Shop*, *FAQ* and *G7-CinemaTicketBot*, which have long conversations and many training phrases, we reduced the testing execution time by letting Botium randomly choose two training phrases per intent (instead of testing with all training phrases). Overall, we created between 36 and 24684 test cases per chatbot.

As an illustration, Fig. 18 shows the schema of a Botium test. It comprises a set of *convo files* describing the expected interactions between the user (label *me*) and the chatbot (label *bot*). The specific user utterances at each interaction step are defined in an *utterance file*. In the figure, the utterances in file *order.drink* use variables like $flavour and $drink (called *scripting memory*), and their values are defined in separate *variables files*. Finally, the expected chatbot outputs at each step are defined in another file (e.g., *order.drink-output0*). Botium executes each convo once per user utterance and variable value. For our experiment, we created convos, utterances and scripting memory values covering all intents, flows, training phrases, chatbot outputs, and entity values defined within the chatbots (Cañizares et al., 2024a).

As the last step, we executed the tests on the generated chatbots. Table 9 shows the results in the column block *"Tests RQ2"*. Column *"#Passed"* (resp. *"#Failed"*) of this block shows the number of tests the generated chatbots pass (resp. fail). Most chatbots (18 out of 20) pass all tests. However, chatbots *RasaProject_Docker* and *santoshkumar30* fail 252 and 84 tests, respectively. A close inspection reveals that all the failed tests were in scenarios containing Empty actions. In the first case, these actions correspond to Python code that built the response dynamically on the original chatbot, and in the second case, to code that asked the value of parameters not defined in the NLP configuration files. Interestingly, chatbot *Email-WhatsApp-Integration-Chatbot* did pass all tests despite having three empty actions. Thus, some Empty actions may have no effect on the chatbot behaviour.

The details of the experiments can be found in https://github.com/Conga-dsl/BotiumExperiment.

Answering RQ2. We answer this RQ by stating that Conga was expressive enough to capture all the details of the selected chatbots, except for some Empty actions. This means that Conga captures Dialogflow chatbots, and Rasa chatbots without behaviour defined in Python code, correctly.

8.4. RQ3: To what extent does Conga automate the migration between Rasa and Dialogflow?

To answer this question, we migrate Rasa chatbots to Dialogflow, and vice versa. Then, to validate behaviour preservation, we perform differential testing as depicted in Fig. 19: we create tests for the original chatbot, and execute them on the chatbot generated for the other platform. The differential testing procedure follows again Algorithm 2.

For this experiment, we use the same chatbots and Botium tests described in RQ2 (cf. Section 8.3). Botium tests are agnostic of the tested chatbot technology, and provides connectors for several technologies like Rasa and Dialogflow. This permits executing the tests defined for Dialogflow over Rasa, and vice versa. Additionally, just like in RQ2, we had to adjust the fallback intent threshold of two generated Rasa chatbots (*fulfilment-regex* and *city-streets-trivia*).

The column block *"Tests RQ3"* of Table 9 shows the results of the test execution. Overall, 17 out of 20 generated chatbots pass all the tests, meaning that they were successfully migrated. Chatbots *RasaProject_Docker* and *santoshkumar30* fail the same tests as in RQ2, again due to the presence of Empty actions. Consequently, these chatbots should be manually adjusted after their migration to Dialogflow. Instead, chatbot *Email-WhatsApp-Integration-Chatbot* is migrated successfully to Dialogflow despite defining Empty actions. In addition, chatbot *city-streets-trivia* does not pass 3120 tests when migrated to Rasa. This is because the original Dialogflow chatbot has several fallback intents, but Rasa only permits one. Our platform warns the developer about this issue by applying the validation rule R3 in Table 2.

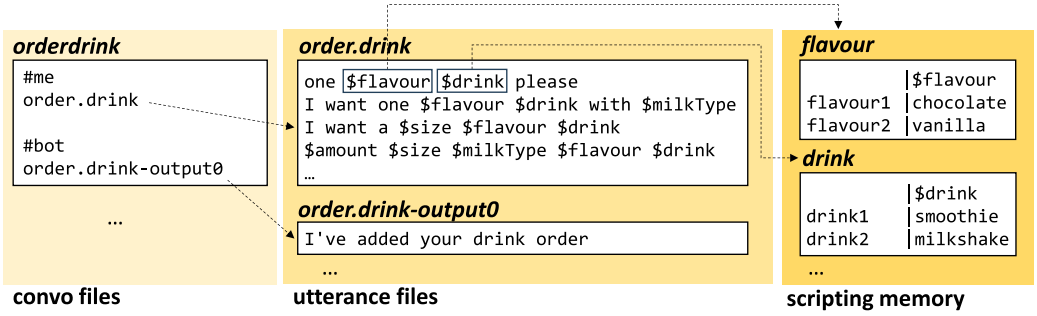

Fig. 18. Schema of a Botium test.

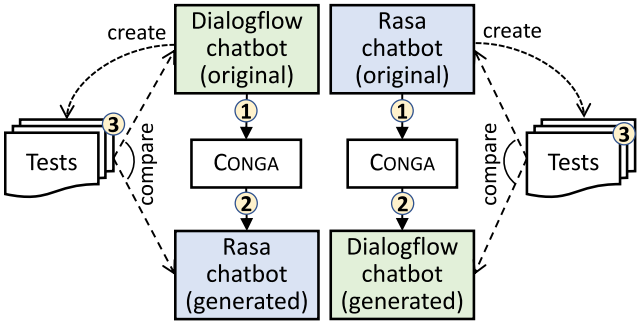

Fig. 19. Research methodology to answer RQ3.

The details of the experiment are available at https://github.com/Conga-dsl/BotiumExperiment.

Answering RQ3. CONGA is able to migrate the NLP behaviour between Rasa and Dialogflow automatically, provided that the parsed chatbots do not have Empty actions (e.g., behaviour in Python), and do not use features that the target technology lacks (e.g., several fallback intents in Rasa). In such cases, CONGA is still able to migrate the chatbots, but manual modifications may be necessary. Likewise, when migrating to Rasa, the default fallback intent threshold works for most cases, though sometimes its manual adjustment may be needed.

8.5. Discussion

Next, we comment on the relevance and implications of the CONGA approach based on the observation of the experiment results.

Chatbot quality assurance. The experiment in Section 8.2 found problems in most chatbots of the dataset. CONGA can help to improve their overall quality since it abstracts low-level tool-specific chatbot implementations into neutral designs, over which validation rules can be applied. The rules may capture general quality problems and tool-specific ones.

Neutral chatbot design notation. The experiment in Section 8.3 demonstrated that CONGA is sufficiently expressive to capture the design underlying chatbots built with diverse technologies such as Dialogflow and Rasa. It does so by providing a neutral chatbot design notation that enables the development of generic services, e.g., for validation (as done in this paper) and metrics (Cañizares et al., 2024b) that can be reused across technologies. Furthermore, it supports the direct engineering of chatbots (as a means to express a chatbot design and generate code once an implementation tool is selected), the reverse engineering of chatbots, and chatbot migration.

Chatbot migration. As Section 8.4 showed, CONGA's neutral meta-model enables the migration of chatbots. We posit that migration is especially relevant in the context of chatbots given the rapidly changing ecosystem of chatbot tools. Without a framework like CONGA, manual migration would be required, which is not trivial (cf. Table 8).

LLMs and future-proofing current chatbots. LLMs are revolutionising many disciplines, including software engineering. In the short-term, we may see a trend in building chatbots based on LLMs. Task-oriented, LLM-based chatbots do not need to define training phrases for intents, but prompts. Still, LLM-based chatbots need to inform the parameters to extract from the user utterances, entities, and conversation flows Cuadrado et al. (2024b). We believe that frameworks like CONGA – enabling neutral chatbot designs and facilitating migration – will become even more valuable as the need to migrate intent-based chatbots to LLM-based chatbots arises.

8.6. Threats to validity

We identify the following potential threats to the validity of our results.

External validity. The collected chatbots may not fully represent the chatbots of commercial or production environments. Thus, the generalisability of the findings to real-world applications could be limited. To mitigate this sampling bias, we ensured a diverse selection of chatbots from different sources to increase the representation of real-world scenarios. Moreover, we conducted the experiments on a relatively small set of chatbots (291 chatbots for RQ1, 20 chatbots for RQ2 and RQ3). A larger sample size might yield other results, and would provide a more comprehensive understanding of CONGA's capabilities.

While CONGA has an extensible architecture, it currently supports just Dialogflow and Rasa. This prevents us from generalising our results to all chatbot platforms. We expect to add support for more tools, which would require further experiments to assess if CONGA captures the chatbot designs in those cases. To mitigate this risk, we designed the CONGA meta-model based on the analysis of fifteen widely-used chatbot tools, as described in Section 4.1.

Construct validity. The tests created with Botium might not fully capture the complexity of the chatbots, leading to incomplete or inadequate test coverage. Moreover, the created tests might not

account for all possible interactions and edge cases. To mitigate this threat, we carefully designed the Botium tests to capture as many relevant details as possible, with full coverage of intents, flows and training phrases, and confirmed that all the tests passed on the original chatbot.

Internal validity. Conga's limitations in handling actions defined in Python within Rasa chatbots could impact the accuracy of migration between Rasa and Dialogflow, leading to incomplete behaviour replication. This is captured as Empty actions, which are a signal for developers of a potential information loss.

In conclusion, while our evaluation provides insights into the capabilities of Conga, we acknowledge the potential threats to validity that need to be considered. We have taken measures to address these threats and achieve a robust evaluation. However, future research could focus on expanding the sample size, refining the conversion process, and exploring additional validation techniques to further strengthen the validity of our results.

9. Related work

The popularity of chatbots has led to the appearance of many tools for their construction. We refer to Pérez-Soler et al. (2021) and Pereira and Díaz (2018) for a classification of the features of these tools, and in the following, we revise works that simplify the forward engineering of chatbots (Section 9.1), their reverse engineering and migration (Section 9.2), or provide suggestions or guidelines for chatbot development (Section 9.3). We finish in Section 9.4 with a comparative, feature-based evaluation of the works considered.

9.1. Automated development of chatbots

Given the effort and expertise required to build chatbots, some authors have proposed automating their development. The most typical approach consists in automatically generating chatbots from well-defined sources, domains or tasks. This approach is used in Castaldo et al. (2019) to generate chatbots for data exploration starting from data models; in Vaziri et al. (2017) to generate chatbots from OpenAPI specifications to help identify the right API; in Ed-Douibi et al. (2021) to produce Xatkit chatbots out of open data APIs; in Baena-Pérez et al. (2020) to create Dialogflow chatbots for video game development; in Pérez-Soler et al. (2020a) to synthesise Dialogflow chatbots from meta-models to support conversational queries; in Pérez-Soler et al. (2019) to generate Dialogflow chatbots to instantiate meta-models using an NL syntax; and in Qasse et al. (2023) to build models via a conversational interface and generate code from them. All these works take as input structured artefacts and apply predefined patterns to generate chatbots for a particular technology and purpose (e.g., data query or data exploration). Instead, Conga enables the generation of chatbots for different technologies, and chatbots may target arbitrary domains, tasks and conversation flows. Actually, Conga could serve as a good target for chatbot generation approaches, since then chatbots for specific platforms could be synthesised.

An alternative way to automate chatbot development is to reuse fragments of existing chatbots. The idea of creating personal chatbots by applying a mashup-based approach was initially proposed in Daniel et al. (2018). In de Lacerda and Aguiar (2019), the authors discuss how automation, pre-configuration, and templates can aid newcomers to develop FAQ chatbots. Similarly, a template-based chatbot construction approach is proposed in Avila et al. (2020), where the templates are SPARQL queries for Enterprise Knowledge Graphs. As in the case of chatbot generation, these approaches are not general-purpose, but they produce chatbots for a specific task. Moreover, Conga enables complementary ways of reuse: of the knowledge about the targeted chatbot tool, and of a same chatbot design to implement it for several technologies.

More similar to our proposal, Xatkit (Daniel et al., 2020) (previously known as Jarvis (Daniel et al., 2019)) is a model-driven solution for developing chatbots that relies on 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 supports chatbot migration.

Finally, Baudart et al. (2018) facilitate the construction of chatbots for the Watson platform via an OCaml library. The approach synthesises JSON configuration files, and uses ReactiveML to orchestrate the conversation. This approach is generative, it is limited to Watson, and does not support reverse engineering.

9.2. Reverse engineering and migration of chatbots

Even if many chatbot development tools use similar concepts, their realisation varies (Pérez-Soler et al., 2021). This is problematic as most of the existing solutions lack portability. Given the plethora of available tools, the early stage of many of them, and the rapid evolution of the field, we expect that migration across technologies will be a common need. Moreover, the advent of LLMs, and the rapid advances in this area, will likely increase the need to migrate to these technologies (Zamfirescu-Pereira et al., 2023). However, there is currently hardly any support for chatbot migration. Just a few solutions like Rasa and early versions of Xatkit are open source. Nonetheless, both demand high technical expertise because they are frameworks where chatbots are developed using general-purpose programming languages (Python in Rasa, and Java in Xatkit). Moreover, only Rasa provides support for migration from Dialogflow, though this support is limited and requires manual data conversion and manual intervention in most migration steps.

Regarding academic works, in Báez et al. (2019), the authors envision a reverse engineering process called *botification* to produce a conversational interface for existing web sites. In a similar vein, Chittò et al. (2020) translate websites into chatbots based on HTML annotations and a mediator service. The latter chatbots can be used to improve web browsing accessibility (Pina et al., 2020). Instead, our system supports migration between chatbot platforms.

Matic et al. (2021) propose a microservice architecture for chatbots, which is independent on the natural language understanding (NLU) component used. This is achieved by the use of a neutral meta-model with NLU concepts (like intents and entities), as well as platform-specific meta-models for Dialogflow and Rasa. This architecture permits selecting the desired NLU component for the chatbot. In our case, Conga includes not only concepts of NLU but also of the conversation flow and chatbot actions, and considers recommendations to specific chatbot development tools. Moreover, Conga does not rely on a chatbot execution engine, but on code generators to synthesise code for the target platform.

Overall, there are proposals to botify non-conversational interfaces but, to the best of our knowledge, not to reengineer existing chatbots. This also implies that there are no general proposals for chatbot migration.

9.3. Recommender systems and guidelines for chatbot development

The popularity of chatbots has raised concerns on proper conversational design. For example, IBM's Natural Conversation Framework (Moore and Arar, 2019) proposes conversation patterns (Moore et al., 2020) and conversation design principles (Moore and Arar, 2018; Schegloff, 2007). The latter include guidelines like *recipient design* (i.e., allow multiple conversation paths for different user types), *minimisation* (i.e., use concise chatbot answers), and *repair* (i.e., provide support for clarifications). In this line, Chatbottest (2024) defines

Table 10
Comparative feature-based comparison of related works.

Name	Forward engineering				Quality assurance		Rev.Eng.&Migration	
	Design Notat.	Exec. Mech.	Extens.	Rec.	Autom. Stat. QA	Extens.	Migration	Extens.
Xatkit (Daniel et al., 2020)	Yes	Direct	Channel	No	Syntax	No	No	No
Baudart et al. (2018)	Yes	Transf.	No	No	Syntax	No	No	No
Matic et al. (2021)	Yes	Direct	NLU Channel	No	Syntax	No	No	No
Rasa	No	Direct	NLU Channel	No	Syntax	No	Dialogflow (partial)	No
Dialogflow	No	Direct	Channel	No	Syntax	No	No	No
Chatbottest	–	–	–	–	Manual	No	No	No
Conga	Yes	Transf.	Platform	Yes	20+2+3	Yes	Rasa Dialogflow	Yes

guidelines for identifying chatbot design issues in categories like answering, error management, intelligence, navigation, personality and understanding. However, the burden is on the developer to manually test whether the chatbot fulfils the guidelines.

Some tools like Dialogflow[6] and Rasa[7] document best practices for designing and implementing chatbots. In both cases, the documentation is focused on the implementation within their tool, however, some of their suggestions can be extrapolated to the development of chatbots in general. For example, Dialogflow recommends giving sufficient training phrases, with a minimum of 10 to 20 examples. Both Dialogflow and Rasa agree on the importance of having a fallback intent, on avoiding similar training phrases in different intents to prevent intent confusion, and on defining varied phrases in each intent to avoid the impression of having a very well trained NLU model when the training data is insufficient. Finally, Rasa emphasises the need to use real data obtained from users using a conversation-driven development process, where the NLU model is fed back with messages from users.

In Abdellatif et al. (2020), the authors discuss challenges in chatbot development by analysing posts in StackOverflow. Most posts are about chatbot development, integration, and NLU. Developers consider the posts about building and integrating chatbots more helpful than other topics. In Abdellatif et al. (2022), the authors compare four NLU components (Dialogflow, LUIS, Watson and Rasa) for their use in software engineering chatbots, and provide five recommendations for fine-tuning the NLU models when developing chatbots. Since Conga is a neutral design notation, it includes validations encoding both platform-specific checkings and general best practices (cf. Table 2). Actually, our rule G15 is also mentioned in Abdellatif et al. (2022) ("*Train NLUs with multiple queries per intent*").

Some researchers have identified important aspects to consider in chatbot design, like functional, integration, analytics and quality assurance (Pereira and Díaz, 2018). In Pérez-Soler et al. (2021), we proposed additional technical and managerial factors which could be used for selecting chatbot development tools. Conga uses these latter factors as a basis for their tool recommendation.

Overall, our contribution in this area is two-fold. On the one hand, extensible validators able to detect at design level errors, potential problems and lack of adjustment to best practices. On the other, an extensible recommender that suggests the most suitable target platform given a chatbot design and other factors.

[6] https://cloud.google.com/dialogflow/es/docs/agents-design#agent_design_best_practices.

[7] https://rasa.com/docs/rasa/generating-nlu-data.

9.4. Comparative analytical evaluation

As a summary, Table 10 compares related works along the main focus of Conga, to stress the main novelty of our proposal.

For forward engineering, column #2 shows whether the approach offers a design notation (e.g., Conga, Xatkit), or if chatbots are defined via forms in a GUI (Dialogflow) or using low-level specifications such as code (e.g., Rasa). Column #3 reports the execution mechanism, which can be *direct* (e.g., there is an engine to run the chatbot, like in Rasa, Dialogflow and Xatkit) or by *transformations* to other frameworks (e.g., the approach of Baudart et al. (2018) transforms into Watson). Column #4 analyses the extensibility mechanisms for forward engineering. Rasa permits adding custom policies and tokenizers for the NLU training process, and connectors to chatbot deployment channels (e.g., connecting a chatbot to a new social network). Dialogflow and Xatkit support adding new deployment channels. The approach by Matic et al. (2021) permits the provision of new channels and NLU processors. Compared to these approaches, Conga permits adding new target chatbot platforms via defining code generators. Finally, column #5 indicates if the approach integrates a recommender for choosing a platform. This feature is only found in Conga.

With respect to quality assurance, column #6 looks at the support for automated static quality checks. The support for static quality assurance is scarce in most tools, like Dialogflow or Rasa, which only check syntactically that there are no references to inexistent entities or intents. Instead, their documentation provides best practices and guidelines for chatbot development. Likewise, Chatbottest guidelines require manual assessment. Instead, Conga offers automated static assessment of 20 quality rules at the design level, and permits adding platform-specific quality rules (currently 2 for Dialogflow and 3 for Rasa). As demonstrated in the evaluation, these quality assurance mechanisms enabled finding issues in existing Dialogflow and Rasa chatbots. While none of the revised works offer extensibility for quality assurance, Conga supports adding new validators, as discussed in Section 5.2.

Regarding migration, as discussed in Section 9.2, Rasa supports partial migration from Dialogflow, but it is not fully automated. In contrast, Conga fully automates the migration between Rasa and Dialogflow, and it is the only approach providing mechanisms to extend the migration capabilities on new platforms.

10. Conclusions and future work

In this paper, we have presented a model-driven solution for chatbot development, re-engineering and migration. The approach is funded on a neutral chatbot design language, along with validators, parsers and code generators for specific chatbot construction technologies. Our

approach does not involve execution, but a recommender suggests the most suitable chatbot tool given a chatbot model and further requirements. We have presented tool support for the approach, and reported an evaluation based on the reverse engineering of chatbot implementations in Dialogflow and Rasa, and their subsequent validation and migration, obtaining good results.

In future work, we plan to add support for more chatbot construction tools into Conga, including prompt-based chatbots powered by LLMs (Cuadrado et al., 2024b). As of today, even if task-oriented LLM-based chatbots are built with prompts and not training phrases, their design shares many features with Conga, like intents, parameters, entities and flows. Hence, we deem an extension of Conga for LLM-based chatbots possible. This will open the door to the automated migration from intent-based, task-oriented chatbots to prompt-based ones. Moreover, it will enable hybrid chatbots, with parts working on LLMs, and others (more sensitive) based on trained intents.

We are currently investigating mechanisms for chatbot quality assurance in the form of static metrics for chatbot designs (Cañizares et al., 2024b), and plan to integrate a recommender for chatbot design refactorings and redesigns based on quality criteria. We would also like to provide variability within Conga chatbot designs by means of product lines. We are working on the automated generation of Botium test cases from Conga models (Cañizares et al., 2024a). We also aim at developing a library of runtime services that developers can select and inject into the code generated from the chatbot designs. The library would include services like conversation persistence, analytics, profiling and user feedback collection. While the focus of this paper was on using Conga for quality assurance and migration of *existing* chatbots, Conga is also useful for direct chatbot engineering. In the future, we plan to conduct user studies to evaluate the notation and the tool.

CRediT authorship contribution statement

Sara Pérez-Soler: Writing – review & editing, Validation, Methodology, Conceptualization, Writing – original draft, Software, Investigation. **Esther Guerra:** Writing – original draft, Investigation, Writing – review & editing, Methodology, Conceptualization. **Juan de Lara:** Writing – review & editing, Methodology, Conceptualization, Writing – original draft, Investigation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Work funded by the Spanish MICINN with projects TED2021-129381BC21, PID2021-122270OB-I00, and RED2022-134647-T.

Appendix

This appendix contains the pseudocode and detailed explanations for parsing and generating Dialogflow and Rasa chatbots from/into Conga models.

A.1. Generating Rasa chatbots from Conga

The entry point of the generation is function doGenerate in Algorithm 3. The function first extracts the intents, entities, parameters, actions, final user interactions, fallback intents and fallback actions (lines 2–8). Then, for each language of the Conga chatbot, it generates one Rasa chatbot with the following files:

- A Python file *actions.py* with the action server code (line 10). The code, generated by function actions in Algorithm 4, validates each entity, and defines classes for handling forms to fill the intents' slots, processing the HTTP requests and responses, and defining placeholder code for the empty actions.
- A yaml file *config.yml* with the configuration of the NLU component (line 11 in Algorithm 3, and function config in Algorithm 5). We use default training and policies pipeline configurations, but customise the fallback action policy if there is a fallback intent.
- A yaml file *credentials.yml* with default URLs and credentials for the services that the chatbot uses (line 12 in Algorithm 3).
- A yaml file *domain.yml* declaring the intents, entities, slots, forms, responses, and actions of the Rasa chatbot (line 13 in Algorithm 3). The content of the file, produced by function domain in Algorithm 6, lists each intent name (lines 2–3); each parameter and its type (lines 4–5); each chatbot response (lines 6–22), including the prompts to ask for missing parameter values (or a default one if absent) and the used text, image and button actions; each action definition (lines 23–24); and a Rasa form for each intent with parameters (lines 25–26).
- A yaml file *endpoints.yml* with a default configuration for the endpoints of the chatbot components, like the action server, the tracker store or the event broker (line 14 in Algorithm 3).
- A markdown file *nlu.md* with the chatbot training data (line 15 in Algorithm 3, and function nlu in Algorithm 7). Specifically, the function lists the training phrases of each intent (lines 2–6), as well as the synonyms and regular expressions of each entity literal (lines 9–10).
- A markdown file *stories.md* with the conversation flows (line 16 in Algorithm 3, and function stories in Algorithm 8). The function produces as many Rasa flows as final interactions in Conga's flows. Then, for each flow, it lists all intents along with their forms (if the intent has parameters) and actions.

Algorithm 3 Generate Rasa Chatbots from a Conga Chatbot.

```
1: function doGenerate(bot: Chatbot) : Zip File                    ▷ entry point
2:     intents ← ALL i: Intent IN bot
3:     entities ← ALL e: Entity IN bot
4:     parameters ← ALL p: Parameter IN bot
5:     actions ← ALL a: Action IN bot
6:     leaves ← ALL u: UserInteraction IN bot with u.next = null
7:     fallbackIntent ← FIRST i: Intent WHERE i.fallback = true
8:     fallbackAction ← a: Action of interaction associated to fallbackIntent
9:     for lan: LanguageIntent IN bot.lang do
10:        CREATE actions.py FILE WITH actions(intents, entities, actions, lan)
11:        CREATE config.yml FILE WITH config(fallbackAction, fallbackIntent)
12:        CREATE credentials.yml FILE
13:        CREATE domain.yml FILE WITH domain(intents, parameters, actions, lan)
14:        CREATE endpoints.yml FILE
15:        CREATE nlu.md FILE WITH nlu(intents, entities, lan)
16:        CREATE stories.md FILE WITH stories(leaves)
17:    end for
18:    STORE files in zipFile
19:    return zipFile
20: end function
```

A.2. Generating Dialogflow agents from Conga

The entry point of the generation is function doGenerate in Algorithm 9. The function generates a set of JSON files with the definition of the Dialogflow agent. The function first creates the file *package.json* with the chatbot version (line 2). Then, it creates the file *agent.json* with the agent meta-data and information of the *webhook*, if there is any — that is why function agentJSON receives the first HTTPRequest in the bot (line 3). Then, additional JSON files are created for the entities (lines 4–9). The content of these files, generated by function entityFile, depends on the type of the entity literals (regular expressions, simple entries, or composite). In addition, for each input language of

Algorithm 4 Generate Code of File *actions.py* for Rasa Chatbot.

```
1: function ACTIONS(intents: Intent[*], entities: Entity[*], actions: Action[*], lan: LanguageIntent) : String
2:     action_code ← GENERATE import statements of rasa modules
3:     for e: Entity IN entities do
4:         action_code ← GENERATE validation code for e, using literals of language lan
5:     end for
6:     for i: Intent IN intents do
7:         action_code ← GENERATE subclass of FormAction for handling slots
8:     end for
9:     for a: Action IN actions do
10:        if a is HTTPRequest or HTTPResponse then
11:            action_code ← GENERATE subclass of Action to handle requests
12:        else if a is Empty then
13:            action_code ← GENERATE subclass of Action with placeholder code
14:        end if
15:    end for
16:    return action_code
17: end function
```

Algorithm 5 Generate Code of File *config.yml* for Rasa Chatbot.

```
1: function CONFIG(fallbackAction: Action, fallbackIntent: Intent) : String
2:     config_code ← GENERATE default NLU training pipeline
3:     config_code ← GENERATE default NLU policies pipeline
4:     config_code ← GENERATE fallback policy with fallbackAction if fallbackIntent ≠ null
5:     return config_code
6: end function
```

Algorithm 6 Generate Code of File *domain.yml* for Rasa Chatbot.

```
1: function DOMAIN(intents: Intent[*], parameters: Parameter[*], actions: Action[*], lan: LanguageIntent) : String
2:     domain_code ← GENERATE start of intents section
3:     domain_code ← GENERATE YAML list with each i: Intent IN intents
4:     domain_code ← GENERATE start of entities and slots section
5:     domain_code ← GENERATE YAML list with each p: Parameter IN parameters
6:     domain_code ← GENERATE start of responses section
7:     for p: Parameter IN parameters do
8:         if p.prompts in language lan ≠ null then
9:             domain_code ← GENERATE YAML item with p.prompts in language lan
10:        else
11:            domain_code ← GENERATE YAML item with default prompt for p
12:        end if
13:    end for
14:    for a: Action IN actions do
15:        if a is Text then
16:            domain_code ← GENERATE YAML item with a's text in language lan
17:        else if a is Image then
18:            domain_code ← GENERATE YAML item with a's image URL
19:        else if a is ButtonAction then
20:            domain_code ← GENERATE YAML item with a's button text and action
21:        end if
22:    end for
23:    domain_code ← GENERATE start of actions section
24:    domain_code ← GENERATE YAML list with each a: Action IN actions
25:    domain_code ← GENERATE start of forms section
26:    domain_code ← GENERATE YAML list with each i: Intent IN intents | i.params ≠ null
27:    return domain_code
28: end function
```

Algorithm 7 Generate Code of File *nlu.md* for Rasa Chatbot.

```
1: function NLU(intents: Intent[*], entities: Entity[*], lan: LanguageIntent) : String
2:     for i: Intent IN intents do
3:         nlu_code ← declare intent i
4:         for tp: TrainingPhrase IN i's phrases for language lan do
5:             nlu_code ← text of tp in Rasa format
6:         end for
7:     end for
8:     for e: Entity IN entities do
9:         nlu_code ← declare synonyms for each entry of e in language lan
10:        nlu_code ← declare regex for each RegularExpression entry of e
11:    end for
12:    return nlu_code
13: end function
```

Algorithm 8 Generate Code of File *stories.md* for Rasa Chatbot.

```
1: function STORIES(leaves: UserInteraction[*]) : String
2:     for i: UserInteraction IN leaves do
3:         stories_code ← list of all intents from i to the beginning of the flow, in reverse order
4:     end for
5:     return stories_code
6: end function
```

Algorithm 9 Generate Dialogflow Agent from a CONGA Chatbot.

```
1: function DOGENERATE(bot: Chatbot) : Zip File                    ▷ entry point
2:     CREATE package.json FILE WITH version information
3:     CREATE agent.json FILE WITH agentJSON(bot, first HTTPRequest IN bot)
4:     for entity : Entity IN bot do
5:         CREATE ⟨entity⟩.json FILE WITH entityFile(entity)
6:         for langEntity : LanguageEntity IN entity.langInputs do
7:             CREATE ⟨entity⟩-entries-⟨langEntity⟩.json FILE WITH entriesFile(langEntity)
8:         end for
9:     end for
10:    for each transition IN bot.flows do
11:        createTransitionFiles(transition)
12:    end for
13:    STORE files in zipFile
14:    return zipFile
15: end function
16:
17: function ENTITYFILE(entity : Entity) : String
18:    entity_code ← entity information depending on its type (regex, simple, composite)
19:    return entity_code
20: end function
21:
22: function ENTRIESFILE(langEnt : LanguageEntity) : String
23:    for entry: Entry IN langEnt.entries do
24:        entries_code ← value and synonyms of entry
25:    end for
26:    return entries_code
27: end function
```

the entity, a JSON file with the entries in those languages is produced. Function entriesFile generates the content of such files.

As the last step, function DOGENERATE creates JSON files with the conversation flows (line 10–12). For this purpose, it makes use of function CREATETRANSITIONFILES in Algorithm 10. This creates a JSON file per intent with information of the interaction (function INTENTFILE), and so-called *user says* json files with the intent training phrases for each language (function USERSAYSFILE). Then, the function traverses the subsequent user interactions (lines 6–8), calling itself recursively with those interactions. If several interactions use the same intent, different files will be generated with different names, built using as prefix the names of the previous intents in the flow. These intents are considered different as their input or output contexts are different.

Algorithm 10 Dialogflow Intent Code Generation.

```
1: function CREATETRANSITIONFILES(transition: UserInteraction) : void
2:     CREATE ⟨intent⟩.json FILE WITH intentFile(transition)
3:     for linput : LanguageInput IN transition.intent.langInputs do
4:         CREATE ⟨intent⟩-usersays.json FILE WITH usersaysFile(input)
5:     end for
6:     for t: UserInteraction: transition.next.next do
7:         createTransitionFiles(t)
8:     end for
9: end function
```

Function intentFile in Algorithm 11 returns a JSON dictionary with information of a interaction step. First, it collects all actions (lines 2–6), either from the next bot response in a linear flow (line 3), or from a previous interaction if there is a loop going back (line 5). Then, it stores the name of the associated intent, and generates a unique id for it (lines 7–8). Next, it stores information of the input context activating the intent, and the output context affected by the intent (lines 9–10). This entails searching the parameters of previous and following intents in the flow. Finally, it stores information of the intent parameters (lines 11–13) and actions (lines 14–16).

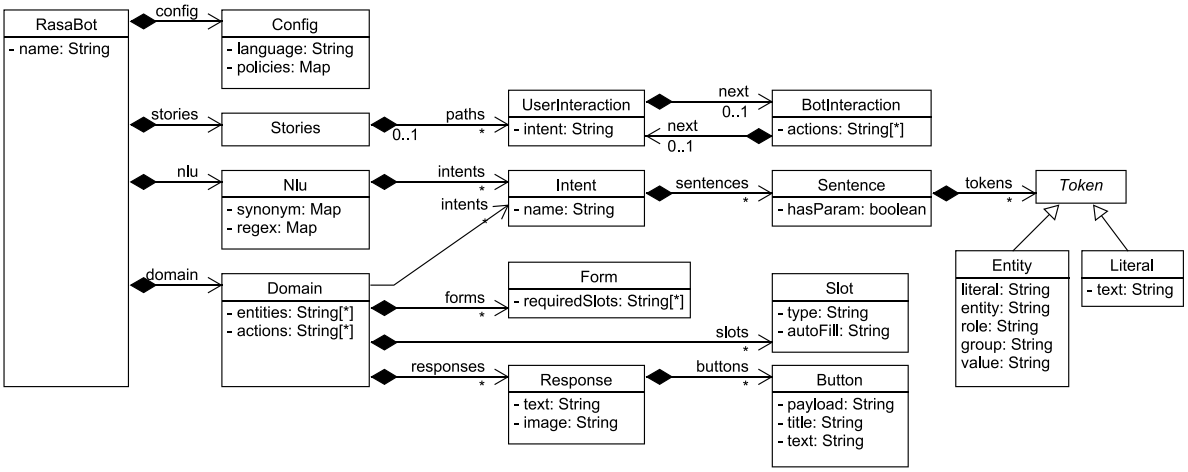

Fig. 20. Rasa meta-model (simplified).

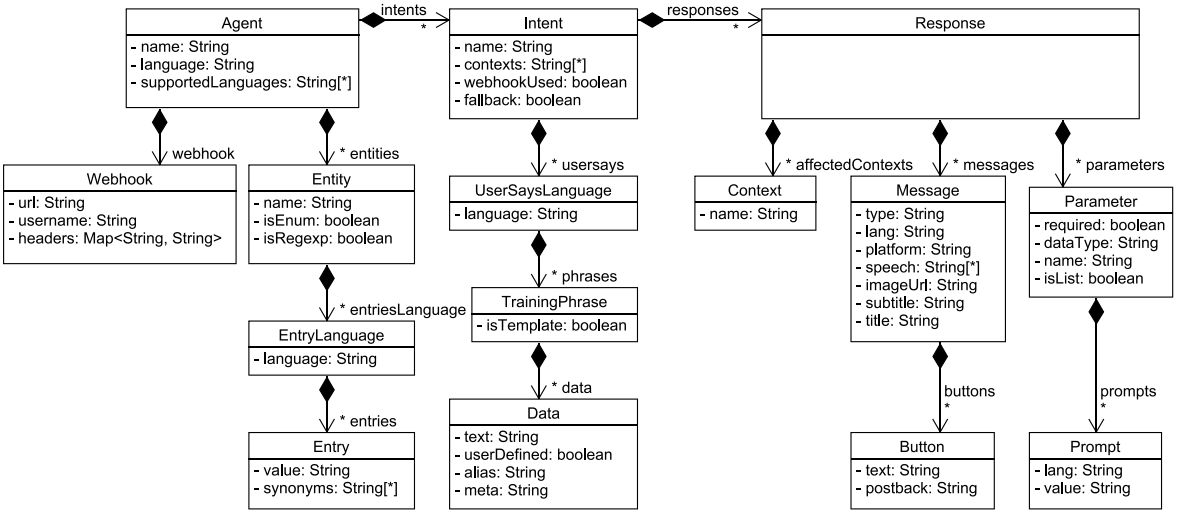

Fig. 21. Dialogflow meta-model (simplified).

Finally, function userSaysFile in Algorithm 11 produces a JSON dictionary with the training phrases for an intent in a certain language. It simply iterates in each phrase, serialising it in Dialogflow's format.

A.3. Parsing Rasa chatbots into Conga

Our parser of Rasa chatbots proceeds in two steps. First, it traverses the yaml and markdown files creating Java classes, in a similar way to a text-to-model transformation. Second, it creates the corresponding Conga objects using the mapping of Table 1.

To facilitate the parsing, we have created the meta-model shown in Fig. 20 (slightly simplified). It permits storing information about the configuration, stories, NLU and domain components of a Rasa chatbot. Specifically, class Config stores the policies pipeline. Class Stories reifies the set of paths (sequences of user and bot interactions) read from file *stories.md*. Class UserInteraction stores the intent being matched, and has a reference to the BotInteraction with its list of associated actions. Class Nlu stores the information read from file *nlu.md*, including the list of intents and their training phrases. Finally, Domain lists the defined intents, the forms with their slots, and the chatbot responses (text, images, buttons).

A.4. Parsing Dialogflow agents into Conga

As for Rasa, the parser of Dialogflow agents proceeds in two steps (model-to-text and model-to-model transformation). First, it traverses the JSON files, creating Java classes conformant to the meta-model shown in Fig. 21 (slightly simplified). In the meta-model, agents define entities, intents, and a webhook for communication with an external service. Entities define entries with synonyms for each supported language. Intents define training phrases for each defined language, responses that include a message and possibly an image and buttons, and the parameters with their prompts. As mentioned in Section 5.1.1, Dialogflow does not support an explicit notion of flow, but the activation of intents is defined via input and output contexts. The former

Algorithm 11 Dialogflow Intent File Code Generation.

```
1: function intentFile(transition: UserInteraction) : String
2:     if transition.next ≠ null then
3:         actions ← transition.next.actions
4:     else
5:         actions ← transition.backTo.previous + transition.backTo.backTo.actions
6:     end if
7:     intent_json ← transition.intent.name
8:     intent_json ← generate unique identifier
9:     intent_json ← input contexts
10:    intent_json ← output contexts
11:    for p : Parameter IN transition.intent.params do
12:        intent_json ← JSON dictionary with p's data
13:    end for
14:    for a: Action IN actions do
15:        intent_json ← JSON dictionary with a's data
16:    end for
17:    return intent_json
18: end function
19:
20: function userSaysFile(linput: LanguageInput) : String
21:    for phrase: linput.phrases do
22:        user_says_json ← phrase text in Dialogflow format
23:    end for
24:    return user_says_json
25: end function
```

identify when the intent is to become active, while the latter set the state after an intent is matched.

Data availability

Conga code and demonstration video: https://saraperezsoler.github.io/CONGA/
Conga tool: http://dimo1.ii.uam.es/CONGA/
Dataset and experiment results for RQ1: https://github.com/Conga-dsl/ValidationSetup
Dataset and experiment results for RQ2 and RQ3: https://github.com/Conga-dsl/BotiumExperiment

References

Abdellatif, A., Badran, K., Costa, D., Shihab, E., 2022. A comparison of natural language understanding platforms for chatbots in software engineering. IEEE Trans. Softw. Eng. 48 (8), 3087–3102.
Abdellatif, A., Costa, D., Badran, K., Abdalkareem, R., Shihab, E., 2020. Challenges in chatbot development: A study of stack overflow posts. In: MSR. ACM, pp. 174–185.
Avila, C.V.S., Franco, W., Maia, J.G.R., Vidal, V.M.P., 2020. CONQUEST: A framework for building template-based IQA chatbots for enterprise knowledge graphs. In: NLDB. In: LNCS, Vol. 12089, Springer, pp. 60–72.
Baena-Pérez, R., Ruiz-Rube, I., Dodero, J.M., Bolivar, M.A., 2020. A framework to create conversational agents for the development of video games by End-Users. In: OLA. In: CCIS, Vol. 1173, Springer, pp. 216–226.
Báez, M., Daniel, F., Casati, F., 2019. Conversational web interaction: Proposal of a dialog-based natural language interaction paradigm for the web. In: CONVERSATIONS. In: LNCS, Vol. 11970, Springer, pp. 94–110.
Baudart, G., Hirzel, M., Mandel, L., Shinnar, A., Siméon, J., 2018. Reactive chatbot programming. In: REBLS@SPLASH. ACM, pp. 21–30.
2024. Botium. https://www.botium.ai/. (Last Access 2025).
Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-Driven Software Engineering in Practice, second ed. In: Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers.
Cañizares, P.C., Avila, D., Pérez-Soler, S., Guerra, E., de Lara, J., 2024a. Coverage-based strategies for the automated synthesis of test scenarios for conversational agents. In: AST. ACM, pp. 23–33.
Cañizares, P.C., López-Morales, J.M., Pérez-Soler, S., Guerra, E., de Lara, J., 2024b. Measuring and clustering heterogeneous chatbot designs. ACM Trans. Softw. Eng. Methodol. 33 (4), 1–43.
Castaldo, N., Daniel, F., Matera, M., Zaccaria, V., 2019. Conversational data exploration. In: ICWE. In: LNCS, Vol. 11496, Springer, pp. 490–497.
2024. Chatbottest. https://chatbottest.com/. (Last Access 2025).
2024. Chatfuel. https://chatfuel.com/. (Last Access 2025).
2024. Chatterbot. https://chatterbot.readthedocs.io/. (Last Access 2025).
Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L., 2021. A survey of compiler testing. ACM Comput. Surv. 53 (1), 4:1–4:36.
Chittò, P., Báez, M., Daniel, F., Benatallah, B., 2020. Automatic generation of chatbots for conversational web browsing. In: ER. In: LNCS, Vol. 12400, Springer, pp. 239–249.
2024. CodeMirror. https://codemirror.net/. (Last Access 2024).
Cuadrado, J.S., Ortiz, R.D.A., Pérez-Soler, S., Cañizares, P.C., Guerra, E., de Lara, J., 2024a. Integrating static quality assurance in CI Chatbot development workflows. IEEE Softw. 41 (5), 60–69.
Cuadrado, J.S., Pérez-Soler, S., Guerra, E., de Lara, J., 2024b. Automating the development of task-oriented LLM-based chatbots. In: ACM Conversational User Interfaces 2024, CUI. ACM, p. 11.
Daniel, G., Cabot, J., Deruelle, L., Derras, M., 2019. Multi-platform chatbot modeling and deployment with the Jarvis framework. In: Proc. CAiSE. In: LNCS, Vol. 11483, Springer, pp. 177–193.
Daniel, G., Cabot, J., Deruelle, L., Derras, M., 2020. Xatkit: A multimodal low-code chatbot development framework. IEEE Access 8, 15332–15346.
Daniel, F., Matera, M., Zaccaria, V., Dell'Orto, A., 2018. Toward truly personal chatbots: On the development of custom conversational assistants. In: SE4COG@ICSE. ACM, pp. 31–36.
de Lacerda, A.R.T., Aguiar, C.S.R., 2019. FLOSS FAQ chatbot project reuse: How to allow nonexperts to develop a chatbot. In: OpenSym. ACM.
Di Ruscio, D., Kolovos, D.S., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M., 2022. Low-code development and model-driven engineering: Two sides of the same coin? Softw. Syst. Model. 21 (2), 437–446.
2024. Dialogflow. https://cloud.google.com/dialogflow/docs. (Last Access 2025).
Ed-Douibi, H., Daniel, G., Cabot, J., 2020. OpenAPI Bot: A chatbot to help you understand REST APIs. In: ICWE. In: LNCS, Vol. 12128, Springer, pp. 538–542.
Ed-Douibi, H., Izquierdo, J.L.C., Daniel, G., Cabot, J., 2021. A model-based chatbot generation approach to converse with open data sources. In: ICWE. In: LNCS, Vol. 12706, Springer, pp. 440–455.
2024. Flow xo. https://flowxo.com/. (Last Access 2025).
2024. Landbot. https://landbot.io/. (Last Access 2025).
2024. Lex. https://aws.amazon.com/en/lex/. (Last Access 2025.
2024. Luis. https://www.luis.ai/. (Last Access 2025).
Matic, R., Kabiljo, M., Zivkovic, M., Cabarkapa, M., 2021. Extensible chatbot architecture using metamodels of natural language understanding. Electronics 10 (18).
McKeeman, W.M., 1998. Differential testing for software. Digit. Tech. J. 10 (1), 100–107.
McTear, M.F., 2020. Conversational AI: Dialogue Systems, Conversational Agents, and Chatbots. In: Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers.
2024. Microsoft bot framework. https://github.com/microsoft/botframework-sdk. (Last Access 2025).
Moore, R.J., Arar, R., 2018. Conversational UX design: An introduction. In: Studies in Conversational UX Design. In: Human-Computer Interaction Series, Springer, pp. 1–16.
Moore, R.J., Arar, R., 2019. Conversational UX design: A practitioner's guide to the natural conversation framework. ACM, New York, NY, USA, ISBN: 9781450363013.
Moore, R.J., Liu, E.Y., Mishra, S., Ren, G., 2020. Design systems for conversational UX. In: CUI. ACM, pp. 45:1–45:4.
Okanović, D., Beck, S., Merz, L., Zorn, C., Merino, L., van Hoorn, A., Beck, F., 2020. Can a chatbot support software engineers with load testing? Approach and experiences. In: ICPE. ACM, pp. 120–129.
2024. ChatGPT. https://openai.com/chatgpt. (Last Access 2025).
2024. Pandorabots. https://home.pandorabots.com/. (Last Access 2025).
Pereira, J., Díaz, O., 2018. Chatbot dimensions that matter: Lessons from the trenches. In: ICWE. In: LNCS, Vol. 10845, Springer, pp. 129–135.
Pérez-Soler, S., Daniel, G., Cabot, J., Guerra, E., de Lara, J., 2020a. Towards automating the synthesis of chatbots for conversational model query. In: EMMSAD. In: LNBIP, Vol. 387, Springer, pp. 257–265.
Pérez-Soler, S., González-Jiménez, M., Guerra, E., de Lara, J., 2019. Towards conversational syntax for Domain-Specific languages using chatbots. J. Object Technol. 18 (2), 5:1–21.
Pérez-Soler, S., Guerra, E., de Lara, J., 2018. Collaborative modeling and group decision making using chatbots in social networks. IEEE Softw. 35 (6), 48–54.
Pérez-Soler, S., Guerra, E., de Lara, J., 2020b. Model-driven chatbot development. In: ER. In: LNCS, Vol. 12400, Springer, pp. 207–222.
Pérez-Soler, S., Guerra, E., de Lara, J., 2021. Creating and migrating chatbots with conga. In: ICSE Companion. IEEE, pp. 37–40.
Pérez-Soler, S., Juárez-Puerta, S., Guerra, E., de Lara, J., 2021. Choosing a chatbot development tool. IEEE Softw. 38 (4), 94–103.
Pina, A., Báez, M., Daniel, F., 2020. Bringing cognitive augmentation to web browsing accessibility. In: ICSOC Workshops. In: LNCS, Vol. 12632, Springer, pp. 395–407.

2024. PlantUML. https://plantuml.com/. (Last Access 2025).
Qasse, I.A., Mishra, S., Hamdaqa, M., 2021. iContractBot: A chatbot for smart contracts' specification and code generation. In: BotSE@ICSE. IEEE, pp. 35–38.
Qasse, I.A., Mishra, S., Jónsson, B.T., Khomh, F., Hamdaqa, M., 2023. Chat2Code: A chatbot for model specification and code generation, the case of smart contracts. In: IEEE International Conference on Software Services Engineering, SSE 2023. IEEE, pp. 50–60.
2024. Rasa. https://rasa.com/. (Last Access 2025).
Ren, R., Castro, J.W., Acuña, S.R., Dieste, O., Acuña, S.T., 2023. Perceived usability of collaborative modeling tools. J. Syst. Softw. (ISSN: 0164-1212) 205, 111807.
Salado-Cid, R., Vallecillo, A., Munir, K., Romero, J.R., 2024. SWEL: A domain-specific language for modeling data-intensive workflows. Bus. Inf. Syst. Eng. 66 (2), 137–160.
Santhanam, S., Hecking, T., Schreiber, A., Wagner, S., 2022. Bots in software engineering: A systematic mapping study. PeerJ Comput. Sci. 8, e866.
Schegloff, E.A., 2007. Sequence Organization in Interaction. Cambridge University Press.
Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2008. EMF: Eclipse Modeling Framework, second ed. Pearson Education.
Tian, Y., Thung, F., Sharma, A., Lo, D., 2017. APIBot: Question answering bot for API documentation. In: ASE. IEEE, pp. 153–158.
Vaziri, M., Mandel, L., Shinnar, A., Siméon, J., Hirzel, M., 2017. Generating chat bots from web API specifications. In: Onward!. ACM, pp. 44–57.
2024. Watson. https://www.ibm.com/cloud/watson-assistant/. (Last Access 2025).
2024. Xtext. http://www.eclipse.org/Xtext/. (Last Access 2025).
Zamfirescu-Pereira, J.D., Wei, H., Xiao, A., Gu, K., Jung, G., Lee, M.G., Hartmann, B., Yang, Q., 2023. Herding AI Cats: Lessons from designing a chatbot by prompting GPT-3. In: DIS. ACM, pp. 2206–2220.
Zhang, N., Huang, Q., Xia, X., Zou, Y., Lo, D., Xing, Z., 2022. Chatbot4QR: Interactive query refinement for technical question retrieval. IEEE Trans. Softw. Eng. 48 (4), 1185–1211.
Zhao, W.X., et al., 2023. A survey of large language models. CoRR, abs/2303.18223 arXiv:2303.18223.