Reusable Abstractions and Patterns for Recognising compositional conversational flows

Sara Bouguelia[1], Hayet Brabra[1], Shayan Zamanirad[2], Boualem Benatallah[2,1], Marcos Baez[1], and Hamamache Kheddouci[1]

[1] LIRIS – University of Claude Bernard Lyon 1, Villeurbanne, France
{sara.bouguelia, hayet.brabra,marcos.baez, hamamache.kheddouci}@univ-lyon1.fr
[2] University of New South Wales (UNSW), Sydney Australia
{shayanz, boualem}@cse.unsw.edu.au

Abstract. Task-oriented conversational bots allow users to access services and perform tasks through natural language conversations. However, integrating these bots and software-enabled services has not kept pace with our ability to deploy individual devices and services. The main drawbacks of current bots and services integration techniques stem from the inherent development and maintenance cost. In addition, existing Natural Language Processing (NLP) techniques automate various tasks but the synthesis of API calls to support broad range of potentially complex user intents is still largely a manual and costly process. In this paper, we propose three types of reusable patterns for recognising compositional conversational flows and therefore automatically support increased complexity and expressivity during the conversation.

Keywords: Conversational Bots · Compositional Conversational Flows · Dialogue Patterns · Nested Intent · Slot value inference

1 Introduction

Task-oriented conversational bots (or simply chatbots) emerged as a paradigm to naturally access services and perform tasks through natural language conversations with software-enabled services and humans. They enable the understanding of user utterances, expressed in natural language, and on fulfilling such needs by invoking the appropriate backend services (e.g., APIs) [22]. However, allowing users to converse naturally with services and perform their tasks effectively is challenging. The main challenge arises from utterance variations in open-end human-bot interactions and the large space of services potentially unknown at development time. Traditional business process and service composition modeling and orchestration techniques are limited to support such conversations because they usually assume a priory expectations of what information and applications will be accessed and how users will explore these sources and services. Limiting conversations to a process model means that we can only support a small fraction of possible conversations [13]. While existing advances in NLP,

rule-based and machine learning (ML) techniques automate various tasks such as intent and slot recognition [5], the synthesis of API calls to support broad range of potentially complex user intents is still largely a manual, ad-hoc and costly process [24]. Our goal is to bridge this gap by dynamically and incrementally synthesizing executable conversation models from natural language conversations. In our previous work, we developed a framework and techniques in this direction [23] including: (i) a word-embedding based API element (e.g. API methods, method parameter) vector space model to support natural language calls to individual APIs [22] and (ii) a hierarchical state machine based model to track and represent human-bot interactions in API-enabled bots [23].

Informed by prior research and literature on conversational systems [5], in this paper, we identify and characterize 3 types of conversation patterns to automatically translate complex user utterances into operations that create composite (nested) states in a bot state machine model: *slot-value-flow*, *nested-intent*, and *API-calls-ordering* patterns. The first pattern allows the bot to resolve a missing value of an intent parameter by extracting it from values of other parameter calls in the conversation history (e.g., a value of output parameter of an already used API call). The second pattern allows the bot to resolve a missing value of an intent parameter by triggering another intent (e.g., a user who wants to schedule a meeting forgets to specify the date. The bot asks for the date and the user responds by a new intent *"Show me my availabilities this week"*). The third pattern allows the bot to map a user intent to a sequence of API calls to satisfy order constraints between two methods of an API (e.g., to fulfil the intent buy a book the bot needs to first call *searchBook* method then *buyBook* method of a *Bookstore* API). These patterns mimics how a developer would have constructed workflows, leveraging conversation knowledge (i.e., slot values and API element vectors), to realise some complex and decomposable user intents. Our approach is motivated by the observation that incorrect inference of conversation flows arises from uncertainty about slot values and relationship between API elements across heterogeneous APIs (e.g., one intent uses *city* as a parameter while another use *location* as a parameter) and complex conversations. More specifically, contributions in this work are summarized as:

- We identify and characterize state machine transformation patterns to support complex user intents. These patterns endow bot platforms with reusable functionality to recognise compositional conversational flows, that would otherwise have to be implemented by bot developers.
- We develop a conversation management service that is augmented with a Context Knowledge Service to support the proposed patterns. This knowledge consists of a graph that represents: (i) slots values relationships and (ii) API methods relationships. It is incrementally derived from conversation utterances and API parameter embeddings.
- Empirical evidence showing the effectiveness of the proposed patterns. The user study showed that these patterns naturally occur when conversing with services, and highlighted the benefits of seamlessly supporting complex user utterances, as perceived by users and confirmed by performance metrics.

In what follows we describe the proposed abstractions, dialog patterns and supporting infrastructure, as well as preliminary evaluation.

2 Related work

A number of techniques have been proposed to build chatbots, including rule-based [2] and probabilistic models [8]. Main platforms [3] such as Chatfuel and FlowXO provide flow-based solutions to develop chatbots with zero coding using UI elements. Research in this context includes the work by Lopez et al. [13], who propose a system that takes a business process model and generates a list of dialog management rules to deploy the chatbot. Other platforms[4] such as DialogFlow, Wit.ai, Amazon Lex and IBM Watson Assistant, on the other hand, provide machine learning (ML) based solutions. In addition to these solutions, a variety of ML models have emerged in research following two common architectures: *pipeline* and *end-to-end*. A *pipeline-based* model is built with a set of components, each responsible for a specific task such as tracking of intent/slot during conversations [6,15], learning next action [17], etc. *End-to-end* models including end-to-end memory networks [25] and sequence-to-sequence models [14] read directly from a user utterance and produces a system action.

We identified a set of main limitations in the works above: First, rules-based approaches lack flexibility and require considerable development effort. Second, the use of existing probabilistic approaches and ML models such as memory networks becomes prohibitive due to the need for collecting huge and high quality training data. Third, flow-based approaches require the explicit definition of workflow, which is clearly unrealistic in large scale and evolving environments. Furthermore, while ML approaches and platforms provide sophisticated support in term of intent/entities recognition and state tracking, they still far to handle conversations as either structured or unstructured processes. This is because they do not yet automatically support complex and decomposable user intents, where handling of intent requires information that is resulted from other intents either already processed or need to be. In addition, handling conversations as processes requires an advanced understanding of conversation context towards natural and straightforward dialogue experiences.

Similar to our approach, some advanced techniques like DEVY [3] and Lu et al. [6] focus on more understanding of context especially by tracking required slots values from conversations history. However, since these graphs are derived only from conversation utterances they do not consider the knowledge of the heterogeneous APIs being used to converse with a wide variety of enabled processes. This aspect is crucial to perform slot values inference accurately. In addition, these works do not propose any pattern that automates the identification of composite conversation flows. While existing co-reference techniques [9] can be used to support slot-value-flow pattern, again such techniques do not employ

[3] Chatfuel: `https://chatfuel.com/`;FlowXO: `https://flowxo.com/`

[4] DialogFlow: `https://dialogflow.com/`, Wit.ai: `https://wit.ai/`, Amazon Lex: `https://aws.amazon.com/lex/`, IBM Watson `https://www.ibm.com/watson/`

API knowledge. Systems like Ava [12] and PLOW[1] can support slot-value-flow pattern in conversations, but by hardcoding variables that refer to values from previous tasks. They also do not provide any automated support for nested-intent and API-calls-ordering. IRIS [7], on the other hand, can enable atomic identification of nested-intent and slot-value-flow patterns, but only to accomplish complex tasks in the data science domain. In addition, such automation is supported by dedicating an API that dynamically adds variables as the conversation progresses to save the result of each dialog task for future use. The key contribution of our approach over these works is greater automation, by enabling the automatic support of conversation patterns through a context knowledge graph that is incrementally derived from conversation history and API knowledge.

3 Overview

A conversation is mainly a sequence of user utterances and bot responses (refers to Figure 1). In addition, studies on human-bot dialogue patterns [10] reveal that conversations are multi-turn (e.g., in Figure 1 there are two turns to accomplish *BookTaxi* intent) and multi-intent meaning that during a conversation user's intent continuously changes as shown in Figure 1. In order to support multi-turn multi-intent conversations, we proposed in our previous work [23] a conversational model that leverages Hierarchical State Machines (HSMs) [21]. HSMs allow to reduce complexity that may be caused by the number of states that are needed to specify interactions between users, chatbots and services.

Inspired by existing workflow management systems and linguistic theory, in this paper, we propose to support greater complexity and expressiveness during conversations by identifying 3 types of dialogue patterns (i.e., *slot-value-flow*, *nested-intent*, and *API-calls ordering*) to realise some complex and decomposable user intents. These transformation patterns along with the conversational state machine model [23] allows to drive incrementally the workflow that steers the conversation with users. We also develop a conversation manager service that aims at initiating and controlling conversations by using a set of services to communicate with users, manage the hierarchical state machine, and invoke APIs. This conversation manager service is augmented with a context knowledge graph to support the proposed transformation patterns. In this section, we give a brief overview of the conversational state machine then, we explain how the conversation manager service manages the conversation flow.

Conversational State Machine. It contains a set of states called *"intent-states"* representing user intents (e.g., *FindRestaurant*), their slots (e.g., *city, food*) and actions such as API invocations (e.g., call *SearchBusinesses* method) to realise them. Inside each intent-state there are states that represent situations that a bot may occupy in a given conversation (e.g., a bot-to-user question to resolve the value of a missing intent slot). Transitions between intent-states automatically trigger actions to perform desired intent fulfillment operations.

Conversation Manager. Figure 2 presents the architecture of the conversation manager service where new main services to support the proposed patterns

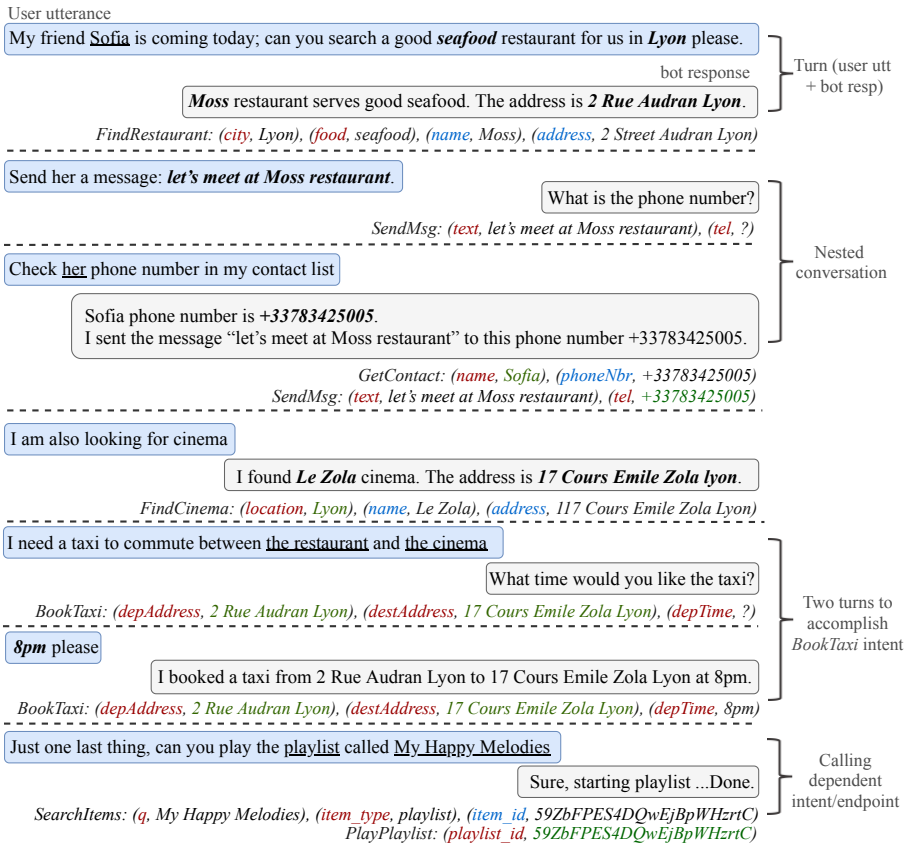

Fig. 1. Example of multi-turn multi-intent conversation. After each turn we illustrate the intent and its set of (slot, value) pairs. The red slots are required input slots, the blue slots are output slots, and the green values are inferred values.

have a red border. The tracker represents the core service that coordinates the information flow in the conversation. When the chatbot receives a new utterance from the user, the *Natural Language Understanding (NLU)* service extracts user intent and (slot, value) pairs from this utterance and sends them to the tracker.

The main objective of the *Dialogue Pattern Recogniser (DPR)* service is to identify compositional conversations (i.e., complex user utterances) and automatically transform them into operations that generate states and transitions in the conversational state machine. In other words, this service checks whether the current utterance is related to a decomposable user intent that involve nested-intent (e.g., *SendMsg* intent involve the fulfillment of *GetContact* nested-intent), or API-calls ordering (e.g., *PlayPlaylist* intent depends on *SearchItems* intent), or a slot-value-flow inference (e.g., infer the value of *Cinema-location* from *Restaurant-city* value to fulfill *FindCinema* intent).

A good understanding of the context is required to correctly infer missing slots' values and identify API methods ordering constraints. Therefore, we intro-

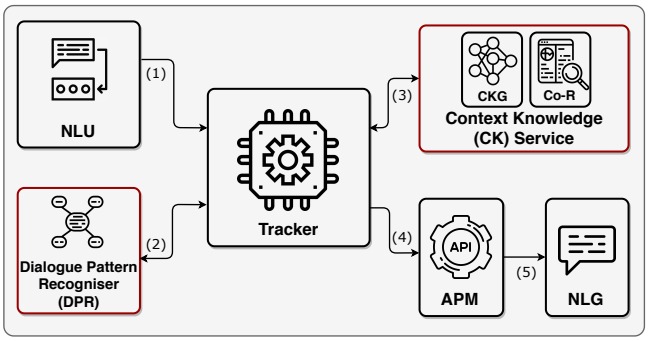

Fig. 2. Conversation Manager Architecture. **(1)** Extract intent and (slot, value) pairs. **(2)** Generate State Machines (SM) operations. **(3)** Infer slot value/Get call ordering of API methods. **(4)** Invoke API method. **(5)** Generate human-like response.

duce the *Context Knowledge (CK)* service that leverages co-reference techniques augmented with a Context Knowledge Graph (CKG) representing slot-value and API methods relationships. This service allows the chatbot to infer slots' values and get call ordering of API methods. Once the tracker collects all required information for the current intent-state, it calls the *API Manager* (APM), which maps the intent-state and (slot, value) pairs to an API method invocation. Finally, the *NLG* service produces a human-like response based on the *APM* output. In the next sections, we describe in detail the *DPR* and the *CK* services.

4 Dialogue Pattern Recogniser

In this section, for each of the three patterns *slot-value-flow*, *Nested-intent* and *API-calls ordering*, we give (i) a description and (ii) an example of how the pattern can automatically recognise compositional conversation flows and transform them into operations in the conversational State Machine (SM).

4.1 Slot-value-flow pattern

Description. The *slot-value-flow* pattern is a known phenomenon in linguistic theory called Anaphora [16]. Anaphora is the use of an expression whose interpretation depends upon another expression mentioned in the conversation history (e.g., in Figure 1, the underlined pronoun *"her"* refers to the entity *"Sofia"*). For chatbots, these expressions are slots' values of previous fulfilled intents that can be reused by the missing slots' values of the current intent.

Example. Figure 3 illustrates an example of how the *slot-value-flow* pattern can be supported in a conversational state machine. Considering the user request *"I need a taxi to commute between the restaurant and the cinema"* in Figure 1, the chatbot detects three missing slots' values (*depAddress*, *destAddress*, *depTime*) in the intent *BookTaxi* (1)(2). The *DPR* service adds a call to a context state to infer the missing slots' values (3). Here the chatbot leverages on the *CK*

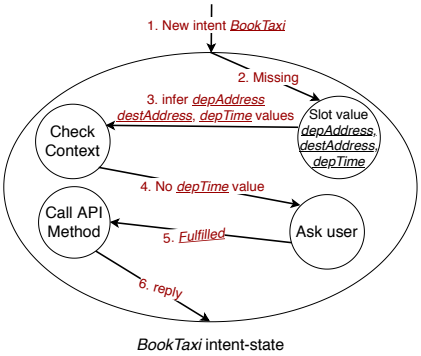

Fig. 3. Slot-value-flow. The SM operations related to BookTaxi intent fulfilment.

service to infer the missing values from previous fulfilled intents (e.g., it infers the value of *Taxi-depAddress* from *Restaurant-address* value). If a missing slot value cannot be inferred (*depTime*), the *DPR* service creates a *"Ask User"* state and the bot asks the user ''*What time would you like the taxi?"* (4). The user answers *"8 pm please"*. Once all required slots for *BookTaxi* intent are fulfilled, the *DPR* invokes the corresponding API method (5) and the bot responds to the user (6).

4.2 Nested-intent pattern

Description. The *nested-intent* pattern is inspired from linguistic theory. In daily life, people have the capability of using nesting conversations [7]. When a friend says *"For when should I book the restaurant?"*, we might respond *"The day Marcos gets back from Milan"*. To automatically translate this linguistic pattern to workflow pattern the *DPR* needs to recognise the *nested-intent state*. There is a nested-intent state when the user wants to accomplish an intent but instead of giving the required slot value, she/he gives another utterance related to another intent that will return the required value to fulfill the parent intent.

Example. Figure 4 illustrates an example of how the nested-intent pattern can be supported in a conversational state machine. Considering the user request in Figure 1 *"Send her a message: let's meet at Moss restaurant."*, the chatbot detects one missing slot value *tel* in the new intent *SendMsg* (1)(2). The *DPR* service adds a call to a context state to infer *tel* value (3). If the value cannot be inferred so the *DPR* service creates the *"Ask User"* state and the bot asks the user *"What is the phone number?"* (4). The user replies by a new utterance *"Check her phone number in my contact list"* related to a new intent *GetContact* (5). Based on the *CK* service the *DPR* can identify that the output slot of *GetContact* has a similar type as the missing slot *tel* (i.e., both represent phone number) therefore there is a high probability that *GetContact* is a nested-intent. the *DPR* creates the nested-intent state *GetContact* (6), gets the required value (*tel* value) and comes back to the parent intent *SendMsg* (7) to call the corresponding API method (8) and respond to the user (9).

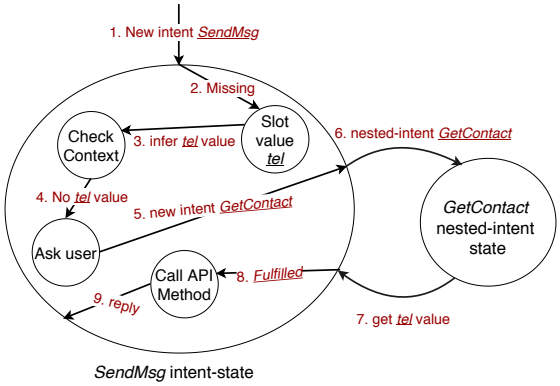

Fig. 4. Nested-intent pattern. The SM operations related to sendMsg intent fulfilment.

4.3 API-calls ordering pattern

Description. We identify a completely new pattern called *"API-calls ordering"* pattern. This pattern is related to REST API design patterns that ensure the discoverability of resources and the ability to access data they refer to [18]. From REST API design perspective, there are some methods that require an API generated string, called *"id"*, as an input parameter to access the needed data. This *id* is an output of another method in the same API.

For example, in *Spotify API*, *SearchItems* method returns an item Spotify Catalog information (e.g., owner, Spotify id, etc.) given the item type (e.g., playlist, albums) and a keyword. On the other hand, *PlayPlaylist* is another *Spotify API* method that requires the returned *Spotify id* to play the corresponding playlist. When the user says *"Play the playlist called My Happy Melodies"*, two scenarios are possible from user perspective. **Scenario 1.** The bot asks the user *"What is the playlist id?"*, but it is unlikely for her/him to know the *id* value because it is a *Spotify API* generated string. To get this *id* the user needs to know that it can be obtained from *SearchItems* method otherwise she/he will not be able to fulfill *PlayPlaylist* intent. She/He says to the bot *"Search for the playlist named 'My Happy Melodies' on Spotify Catalog"*. The bot fulfills the *SearchItems* intent and returns the *id* value. The user asks again *"Can you start the playlist with this id 59ZbFPES4DQw..."*. In this scenario, the user is forced to adapt to the technology. **Scenario 2.** The bot responses directly to the user saying *"Sure, starting playlist... Done."*. To support this scenario a bot developer needs to implement an intermediately endpoint that combines *SearchItems* and *PlayPlaylist* endpoints. The implementation of new endpoints could grow exponentially if the bot developer have to account for all endpoints pairs. In the following, we explain through the same example how this *API-calls ordering* pattern can be automatically supported in state machines.

Example. In Figure 5, when the user says *"Play the playlist called My Happy Melodies"*, the *DPR* creates a new intent-state called *PlayPlaylist* (1). It detects that the *id* value is missing (2) and adds a call to a context state (3). Using *CK* service, the *DPR* recognises that *PlayPlaylist* intent depends on *SearchItems*

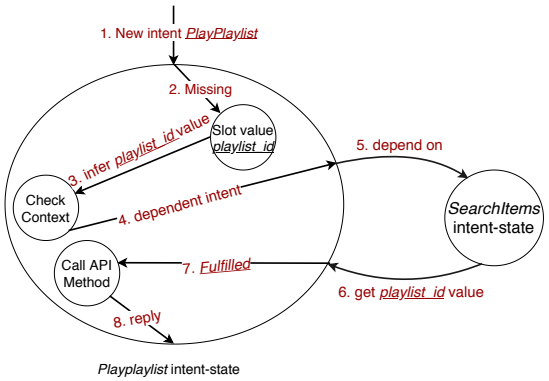

Fig. 5. API-calls ordering. The SM operations related to PlayPlaylist intent fulfilment.

intent (4). In consequence, it creates the *SearchItems* intent-state (5) and fulfills it to get the *id* value (6). Then, the *DPR* comes back to *PlayPlaylist* to call the corresponding API (7) method and respond to the user (8).

5 Context Knowledge Service

Context can be defined as any information that can be leveraged from previous turns or external knowledge [11]. Maintaining the context is necessary in chatbots as it allows to keep continuity in the dialogue and avoid repetition, making interactions more natural [11]. However inferring information from the conversation context is challenging due to multi-turn multi-intent conversations and heterogeneous APIs. There are several parameters among multiple APIs methods that can share all or some of their values during the conversation [19].

To tackle this challenge, we propose the *Context Knowledge (CK)* Service that aims at accumulating a part of the conversation context. The *CK* service uses a co-reference system to resolve a potentially referenced slot value (e.g., *Contact-name* value is referenced by the pronoun *"her"* in the utterance *"Check her phone number"*). This system links together mentions that relates to entities given the previous turns. In our approach, we choose to use *Neuralcoref* [9], a state-of-the-art coreference resolution system based on neural networks.

In some cases, the co-reference system is not able to infer the correct slot value. For example in Figure 1, when user says *"I need a taxi to commute between the restaurant and the cinema"*, the co-reference will replace the mention *"the restaurant"* by the entity *"Moss restaurant"* which is a wrong value for the *Taxi-depAddress* slot because it should be an address. To have more accurate slot value inference, we augment the co-reference system with a *Context Knowledge Graph (CKG)* that represents: (i) slots values relationships and (ii) API methods relationships including call-ordering constraints. Figure 6 shows the *CKG* instance related to the conversation in Figure 1.

Context Knowledge Graph We denote the graph as $G = (N, E)$, where N and E are the set of nodes and the set of edges respectively. In this graph there are 4 node types: entity (e.g., *Cinema*), attribute/slot (e.g., *city*), entity

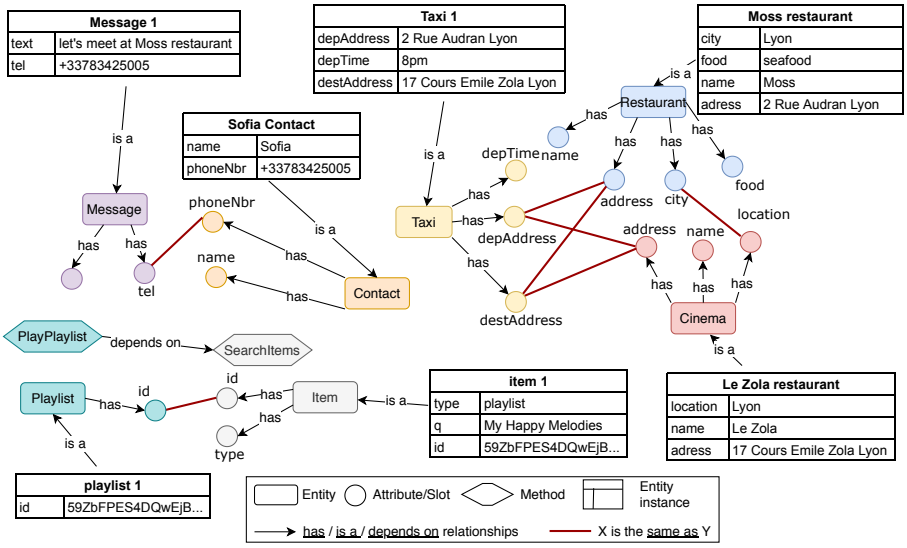

Fig. 6. Example of a context knowledge graph related to the conversation in Figure 1.

instance (e.g. *Le Zola cinema*), and methods (e.g., *PlayPlaylist*). There are 4 edges types: *"has"*, *"is a"*, *"depends on"*, and *"same as"*. The relationship *"has"* is generated between an entity and an attribute based on a predefined ontology. The relationship *"is a"* is generated between an instance and its entity (i.e., node *is an* instance of). Both *entity instances* and *is a* edges are incrementally derived from the conversation. The relationship *"depends on"* denotes that a method *depends on* another method to get the required *id* value. The computation of this *"depends on"* edge can be done from API reference documentation using the open information extraction methods proposed by [20]. The *"same as"* edge is generated between two similar slots (e.g., in Figure 6 there is an edge between *Cinema-address* and *Taxi-depAddress*, because the taxi departure may be the cinema's address). The computation of the *"same as"* edge is based on the semantic similarity between the *slots vectors* embedding. A *slot vector* is the average of the vectors values embedding. These values represent the possible values that each slot can take. We acquire such possible values by querying API-KG Web Service[5] that returns a list of values for a given API method parameter. Then, we compute the cosine similarity between each slot pair and compare this similarity with a threshold predefined by the bot developer. If the similarity is greater than the threshold, an edge is created between the two slots.

Slot Value Inference Figure 7 shows an example of the main steps to infer a missing slots' values {*depAddress, destAddress, depTime*} from a given utterance

[5] https://apikg.ap.ngrok.io/api/docs/

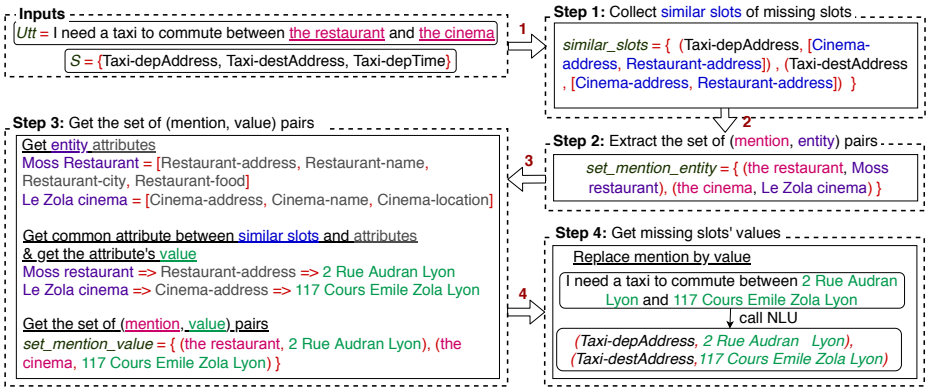

Fig. 7. Example of inferring slots' values.

"I need a taxi to commute between the restaurant and the cinema" based on the *CKG*.

Step 1. We collect the set of similar slots based on the *"same as"* edges. If a missing slot value has only one similar slot (e.g., *Cinema-location* has only *Restaurant-city*), we reuse directly the similar slot's value. *Taxi-depTime* is not considered in next steps because it doesn't have any similar slot.

Step 2. The co-reference system extracts (mention, entity) pairs from *Utt*.

Step 3. For each entity returned by the co-reference system, we get its list of attributes. Next, we get the common attribute using intersection between the list of similar slots and the list of entity attributes. Finally, we get the value of each common attribute and link it to the corresponding mention.

Step 4. Lastly, each mention in *Utt* is replaced by its corresponding value. The utterance *Utt* is sent to the NLU service to extract the missing slots' values.

6 Evaluation

In this section we describe a study aiming at understanding the need, benefits, and effectiveness of supporting the proposed patterns. We investigate whether the proposed patterns naturally occur when conversing with services, perform a comparative analysis with alternative approaches focusing on the user experience, to then analyse the support provided based on performance metrics.

6.1 Methods

Experimental design. Participants were recruited from the extended network of contacts of the authors. Invitations were sent via email, asking for volunteers,

resulting in a total of 12 participants. We followed a within-subjects design[6] to evaluate the proposed dialog patterns and supporting services. Participants were tasked with interacting with three different chatbots, which were developed to capture the following experimental conditions:

- *DF-Baseline* : The baseline implements the standard conversational management support of traditional chatbot development platforms. It is developed using the underlying techniques of DialogFlow, including the DF NLU model, conversational model, and the Input-Output context mechanism.
- *SM-NeuralCoref* : it relies on the State Machine conversational model but supporting only the NeuralCoref model [9]. The aim of this setup is to emphasize the need for the Context Knowledge Graph.
- *SM-Patterns* : it includes all of our services to support the new proposed conversation patterns and relies on the State Machine conversational model.

Besides the differences highlighted above, all three chatbots were built on the same foundation. They supported 15 intents collected from the DSTC8 dataset [15]. For all three chatbots, we use DialogFlow NLU service as NLU model because it is one of the most complete NLU models [4] to train chatbots.

We devised three main tasks, each comprising representative scenarios that catered to the proposed dialog patterns. Task 1, on the slot-value-flow pattern (T1), required participants to plan a day program by interacting with services that would benefit from leveraging the ongoing context of the conversation (e.g., reusing same locations or date). Task 2, on the nested-intent pattern (T2), asked participants to schedule a doctor's appointment on the first available spot. The dependency between the involved services favored the use of a nested pattern. Task 3, on the API-calls ordering pattern (T3), invited users to look for a restaurant with good ratings, requiring them to interact with services (search, reviews) linked by an ID. It is important to note that each scenario suggested the need for relevant services without imposing any specific conversation style or order.

Procedure. The study was conducted online with the support of an online form aggregating all the instructions. Before starting, participants provided their consent to participate and for their interactions with the chatbots to be recorded. After providing background information, participants then proceeded to perform the tasks with the three chatbots, in a randomised order to avoid positional bias. For each task, participants were asked to describe the pros and cons of their experience with each chatbot, and to specify which one provided the better experience and why. The duration of the experiment was between 45-90 minutes.

Data processing and analysis. We performed a qualitative and quantitative analysis of the experience with each chatbot. We performed a thematic analysis of the open-ended participant feedback so as to identify emerging themes in their experience with the chatbots, and better characterise the reasons behind their preferred design. The conversation logs were also analysed to i) understand if participants naturally engage in conversations that leverage the proposed dialog

[6]Study materials and in-depth results available at https://tinyurl.com/25ad8jv6

Table 1. Evaluation of chatbots according to performance metrics. Arrows indicate lower values better (↓) and higher better (↑), and bold face best performance. Percentages denote the relative performance with respect to the reference (optimal) scenario.

Metric	Reference	DF-Baseline	SM-Patterns	SM-NeuralCoref
M1 ↓ (TURNS)	**8,42**	9,92 (18 %)	**8,67 (3 %)**	10,83 (29%)
M2 ↓ (PROMPTS)	**4,25**	5,58 (31 %)	**4,42 (4 %)**	6,33 (49 %)
M3 ↑ (SLOTS)	**3,33**	1,33 (-60 %)	**3,17 (-5 %)**	0,08 (-98 %)

patterns, and ii) assess the performance of the chatbots. We analyse the performance in relation to the optimal reference scenario (e.g., the most efficient scenario for the conversation style adopted by the participant) by considering the following metrics: number of (M1) conversation turns, (M2) prompts asking for missing slot values, and (M3) missing slot values correctly inferred.

6.2 Results

T1. Slot-value-flow pattern. The large majority of participants (9/12) reported having a superior experience when interacting with the *SM-Patterns* chatbot as compared to the alternatives. The qualitative analysis of participant feedback revealed two main reasons behind this preference. The dominant theme was the **efficiency of interactions** (9 participants), with participants expressing the *SM-Patterns* chatbot being *"quicker in getting an answer"* (P12) and being able to correctly infer missing values (e.g., *"I liked that it correctly understood my destination and I didn't have to input the address [from a previous turn]"*, P10). Another salient theme was the ability to enable more **natural conversations** (6 participants), with participants explicitly stating the *"experience of the conversation [being] more natural and human-like"* (P14). Participants also suggested improvements, notably in terms of being transparent (2 participants) about what information the chatbot was inferring from the context.

The analysis of the conversation logs showed that the majority of participants (9 participants) engaged in conversations styles that took full advantage of this pattern, successfully referencing the context at least twice. Interestingly, the participants who showed preference towards the other chatbots engaged in conversation styles that to a lesser degree benefited of the slot-value-flow pattern, and instead formulated utterances that provided actual slot values in the requests (e.g., U: *"I want a taxi to [address]"*) instead of leveraging the context.

The quantitative analysis of chatbot performance (Table 1) confirms the qualitative observations, putting the support by SM-Patterns as the closest to the optimal performance (reference scenario) for the three metrics under evaluation. In contrast, the simple support by DF-Baseline resulted in longer conversations and required more input from the users. Interestingly, SM-NeuralCoref performed the poorest even when supporting co-reference techniques, but this can be attributed to its inability to accurately infer missing slot values (M3).

T2. Nested-intent pattern As in the previous task, the majority of participants expressed their preference for *SM-Patterns* (9/12 participants). The

qualitative analysis of the feedback identified four main themes behind this preference. Participants referred to the chatbot's ability to **keep track of the user goal** (6 participants), stating that when engaging in a nested intent *"[the chatbot] remembered that I wanted to book appointment with a dentist (user goal)"* (P4) while the baseline would *"forget totally [what] I wanted"* (P3). Providing a **natural flow** was another emerging quality attribute (4 participants), with participants expressing that the experienced *"flow felt natural"* (P6) while the baseline would force them to plan ahead. The chatbot was also perceived as **efficient** (5 participants), requiring *"less input for a correct answer"* (P14), while for a few it simply came down to being **effective** (2 participants), i.e., able to complete their task with the conversation styles they engaged in.

An analysis of the conversation logs revealed that most participants (7/12) had naturally described a nesting-intent pattern in their interactions. Looking into the conversation logs of those who expressed preference for the baseline (3 participants) provided further insights. Interestingly 2 of these participants had not actually engaged in a nested-intent pattern, while the one who did had experienced problems in the formulation of the nested intent (i.e., the framing of the nested intent was not recognised by the NLU). This highlights the need for integrating conversation repair strategies into this pattern.

T3. API-calls ordering pattern. All participants (12/12) reported having a better experience with the *SM-Patterns* chatbot. Not surprisingly, the majority of participants (8 participants) commented on the ability to **hide technical details** as one of the main reasons for their preference, one participant citing that in the proposed scenario *"it successfully understood that I wanted a review from the selected restaurant without asking for the business ID"* (P7), whereas the technical details of the service as exposed by the baseline chatbot made it *"difficult to understand for someone who doesn't know what that means"* (P3). Providing a **smooth conversation flow** was another theme that emerged from the feedback on *SM-Patterns* , with participants mentioning that in comparison, interacting with the baseline chatbot felt like being *"caught in a loop"* (P8). Some participants summarised the positive experience by simply stating that the chatbot was **effective**, working correctly or as expected.

The analysis of conversation logs showed that all but one participant (who deviated from the proposed scenario) described interactions that benefited from the API-calls ordering pattern. What this tells us is this pattern greatly aligns with the conversation styles and expectations of users.

7 Conclusions and Future Work

In this paper, we identified and characterized 3 types of dialog patterns that endow bot platforms with reusable functionality to recognise compositional conversational flows and reduce the development complexity. Our work also comes with its own limitations and space for possible improvements. While we provide empirical support for the proposed dialog patterns, the evaluation is still limited in the number of participants, and so we plan to run larger scale evaluations. In

addition, in this work we only focused on Intent-SingleAPI interaction pattern, and we plan to investigate other patterns such as Intent-CompositeAPI, where user utterance may involve more than one intent.

References

1. Allen, J., et al.: Plow: A collaborative task learning agent. AAAI (2007)
2. Banchs, R.E., Jiang, R., Kim, S., Niswar, A., Yeo, K.H.: AIDA: Artificial intelligent dialogue agent. In: Proc. SIGDIAL 2013. pp. 145–147 (2013)
3. Bradley, N., Fritz, T., Holmes, R.: Context-aware conversational developer assistants. In: 40th International Conference on Software Engineering (ICSE) (2018)
4. Canonico, M., De Russis, L.: A comparison and critique of natural language understanding tools. Cloud Computing **2018**, 120 (2018)
5. Chen, H., Liu, X., Yin, D., Tang, J.: A Survey on Dialogue Systems: Recent Advances and New Frontiers (1)
6. Chen, L., al.: Schema-guided multi-domain dialogue state tracking with graph attention neural networks. Proc. AAAI 2020 **34**, 7521–7528 (2020)
7. Fast, E., et al.: Iris: A conversational agent for complex tasks. CHI '18 (2018)
8. Henderson, M.S.: Discriminative methods for statistical spoken dialogue systems. Ph.D. thesis, University of Cambridge (2015)
9. Hugging-Face: Fast coreference resolution in spacy with neural networks, `https://spacy.io/universe/project/neuralcoref`, Last accessed on 2020-11-15
10. Hutchby, I., Wooffitt, R.: Conversation analysis. Polity (2008)
11. Jain, M., Kota, R., Kumar, P., Patel, S.N.: Convey: Exploring the use of a context view for chatbots. In: Proc. CHI 2018. pp. 1–6 (2018)
12. John, R.J.L., Potti, N., Patel, J.M.: Ava: From data to insights through conversations. In: 8th Biennial Conference on Innovative Data Systems Research (2017)
13. López, A., Sànchez-Ferreres, J., Carmona, J., Padró, L.: From process models to chatbots. In: Giorgini, P., Weber, B. (eds.) CAiSE '19 (2019)
14. Manning, C.D., Eric, M.: A copy-augmented sequence-to-sequence architecture gives good performance on task-oriented dialogue. In: EACL (2017)
15. Rastogi, A., et al.: Towards Scalable Multi-domain Conversational Agents: The Schema-Guided Dialogue Dataset. arXiv e-prints arXiv:1909.05855 (Sep 2019)
16. Reinhart, T.M.: The syntactic domain of anaphora. Ph.D. thesis, MIT (1976)
17. Su, P., el.: Continuously learning neural dialogue management. CoRR (2016)
18. Thomas, E., et al.: Soa with rest-principles, patterns and constraints for building enterprise solutions with rest. The Prentice Hall service technology series (2013)
19. Wu, C.S., et al.: Transferable multi-domain state generator for task-oriented dialogue systems. arXiv preprint arXiv:1905.08743 (2019)
20. Xiaoxue, R., et al.: Api-misuse detection driven by fine-grained api-constraint knowledge graph (2020)
21. Yannakakis, M.: Hierarchical state machines. In: TCS. pp. 315–330. Springer (2000)
22. Zamanirad, S.: Superimposition of natural language conversations over software enabled services. Ph.D. thesis, University of New South Wales, Australia (2019)
23. Zamanirad, S., et al.: Hierarchical state machine based conversation model and services. Proc. CAiSE 2020
24. Zamanirad, S., et al.: Programming bots by synthesizing natural language expressions into api invocations. In: Proc. ASE 2017. pp. 832–837. IEEE (2017)
25. Zhang, Z., et al.: Memory-augmented dialogue management for task-oriented dialogue systems. ACM Transactions on Information Systems **37**(3), 1–30 (2019)