Context Knowledge-aware Recognition of Composite Intents in Task-oriented Human-Bot Conversations

Sara Bouguelia, Hayet Brabra, Boualem Benatallah, Marcos Baez, Shayan Zamanirad, Hamamache Kheddouci

Context Knowledge-aware Recognition of Composite Intents in Task-oriented Human-Bot Conversations

Sara Bouguelia[1] 0000-0003-4538-0927, Hayet Brabra[1] 0000-0001-7484-2268, Boualem Benatallah[2] 0000-0002-8805-1130, Marcos Baez[1] 0000-0003-1666-2474, Shayan Zamanirad[2] 0000-0002-3371-9631, and Hamamache Kheddouci[1] 0000-0002-5561-6203

[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
{boualem, shayanz}@cse.unsw.edu.au

Abstract. Task-oriented dialogue systems employ third-party APIs to serve end-users via natural language interactions. While existing advances in Natural Language Processing (NLP) and Machine Learning (ML) techniques have produced promising and useful results to recognize user intents, the synthesis of API calls to support a broad range of potentially complex user intents is still largely a manual and costly process. In this paper, we propose a new approach to recognize and realize complex user intents. Our approach relies on a new rule-based technique that leverages both (i) natural language features extracted using existing NLP and ML techniques and (ii) contextual knowledge to capture the different classes of complex intents. We devise a context knowledge service to capture the requisite contextual knowledge.

Keywords: Task-oriented Conversational Bots · Complex Intent Recognition · Context knowledge · Slot value inference

1 Introduction

Task-oriented dialogue systems (or simply bots) use natural language conversations to enable interactions between humans and software-enabled services [3]. In these, fulfilling a user request consists of: (1) understanding the *user utterance* expressed in natural language (e.g., "What is the weather in Paris?"), (2) recognizing the *user intent* (e.g., `GetWeather`), (3) extracting relevant *slot-value* pairs (e.g., (`location, Paris`)), (4) invoking the corresponding API (e.g., `OpenWeatherMap` to get weather condition), and (5) returning a natural language response to the user (e.g., "We have light rain in Paris").

Ideally, the bot should detect intents and infer slot values with the least possible interactions with the user (i.e., the bot asks the user for a missing value only when it cannot infer it from other sources). A key challenge to achieve

this objective is devising robust intent recognition and slot inference despite the potentially ambiguous and complex utterances. An utterance may not always follow a simple conversation pattern, where the bot recognizes a *basic intent* and infers all required slot values from the utterance, as in the previous example.

Natural user conversations can be rich, potentially ambiguous, and express *complex user intents* [6,21]. An intent is complex when its realization requires the bot to break it down into a list of atomic actions and infer potentially missing values from different sources, not directly from the utterance. Given the utterance "Can you book a table for 2 people at Mirazur restaurant for the next public holiday?", the bot should be able to infer the information such as number of people and restaurant from this utterance; however, it also needs to search when will the next holiday be. Failing to support such complex intents can lead to repetitive and less natural interactions affecting the user experience [11].

Existing NLP and ML techniques have produced useful results to recognize basic intents [19]. ML based techniques rely on the availability of massive amounts of annotated data. Using these techniques to recognize complex intents requires laborious, costly and hard to acquire training datasets. In addition, each time a new complex intent is identified, extending or producing a new dataset is needed as well. Therefore, more advanced and flexible techniques that cater for complex intent recognition are needed.

In our previous work, we identified and characterized a set of composite dialog patterns that naturally emerge when conversing with services [2]. In this paper, we focus on the *recognition* of complex intents in human-bot conversations. We take the view that complex intent recognition could be significantly improved by considering composite dialog patterns in addition to basic intent features. We propose an approach that relies on (i) existing NLP and ML techniques to extract natural language features (e.g., basic intents, dialog acts) and (ii) a rule-based approach that leverages these features together with contextual knowledge, enabled by composite dialog patterns and other metadata, to define complex intent recognition rules. These rules enable a higher-level of abstraction that offers flexibility for an *extensible* library of composite dialog patterns. When a new complex intent class is identified, a new rule template is added to recognize intents of this class from utterances. This approach requires to capture fairly complex context knowledge in addition to basic intents in order recognize complex intents. Thus, there is a need for advanced context representation and exploitation techniques that go beyond conversation history to include information inference that leverage metadata such as intent and API schemas (e.g, intents, slots, API methods) and relationships between their elements. Our contributions in this work are summarized as:

- We propose a rule-based approach that combines (i) natural language features (ii) composite dialog patterns and contextual knowledge to capture different classes of complex intents in a generic way.
- We propose major extensions to the preliminary context knowledge service (CKS) presented in [2]. These extensions consist of an improved context knowledge model and a set of new services providing the contextual knowledge that is needed for the rules to recognize complex intents.

– Empirical evidence showing the effectiveness and user experience of the CKS and complex intent recognition. The user study showed that endowing bots with the complex intent recognition allow more natural interactions, as perceived by users and confirmed by performance metrics.

This paper is organized as follows: Section 2 describes a scenario and the general architecture; Section 3 details the CKS; Section 4 shows the rules to recognize and realize complex intents; Section 5 presents the experimentation; Section 6 presents the related work; Section 7 presents the conclusion and future work.

2 Scenario and Architecture

Before delving into the main contributions, we first introduce a scenario illustrating the proposition value of the CKS, and the supporting architecture.

2.1 Scenario

Consider a user who wants to plan some activities by conversing with a bot, as shown in Figure 1. The user interacts with the bot using complex intents where there may exist missing values. Existing state-tracking[3] (ST) techniques support these interactions only in some cases and tend to be chatty and prompt users for the missing values. Slot values can be inferred by leveraging different sources:

Infer slot value from conversation history. The composite pattern called *slot-value-flow* allows resolving a missing value of a slot by extracting it from conversation history. Existing ST techniques provide a limited support to this pattern. For example, in utterance #2, they can deduce that the missing value of the slot `cinema-area` is the same as the value of the slot `restaurant-area` and reuse it. However, in utterance #3, they use a *coreference* model to replace the expressions "the cinema" and "the restaurant" with the mentions "UGC cinema" and "LaGoulette restaurant", respectively, which are wrong values because they should be addresses. This will lead the bot to ask the user to provide the precise addresses values. A bot improved with CKS can infer these values by detecting that the `departure` is more likely related to the `cinema-address` than the `cinema-name` and thus reusing the address value.

Infer slot value by calling another API method. The composite pattern called *nested-method* allows resolving a missing value of a slot by triggering another method. For example, in utterance #4, the user wants to send a message. Instead of giving the recipient's phone number (i.e., `tel`), the user gives an expression that refers to the recipient's name (i.e., "my friend" to refer to "Hayet"). While existing ST techniques can deduce that the expression "my friend" refers to "Hayet", they cannot infer the value of the slot `tel`. A bot improved with CKS can detect that there is an API method having an output similar to the missing slot and therefore invokes this API method to infer the missing value.

---

[3]State-tracking consists of determining user intent and its required slot values.

#1 User: I am looking for a **Tunisian restaurant** in **Paris**
  Bot: LaGoulette restaurant serves Tunisian food. The address is **59 Rue de la verrerie Paris.**
INTENT: *SearchRestaurant: (restaurant-area, Paris), (food, Tunisian restaurant)*
API-METHOD: *Yelp-SearchBusiness: (location, Paris), (term, Tunisian restaurant), (name, LaGoulette), (address, 59 Rue de la verrerie Paris)*

#2 User: I am also looking for a **cinema**
  Bot: I found the **UGC** cinema. The address is **166 Boulevard Macdonald Paris.**
INTENT: *SearchCinema: (cinema-area, ?), (term, cinema)*
API-METHOD: *Yelp-SearchBusiness: (location, **Paris**), (term, cinema), (name, UGC), (address, 166 Boulevard Macdonald Paris)*
                                                └──── *value inferred from history*

#3 User: I am excited to see my friend Hayet! Book a taxi from the cinema to the restaurant **today** at **5 pm**
  Bot: Done! I booked a taxi from **166 Boulevard Macdonald Paris** to **59 Rue de la verrerie Paris today** at **5 pm.**
INTENT: *BookTaxi: (departure, ?), (destination, ?), (date, today), (time, 5 pm)*
API-METHOD: *taxicode-booking: (pickup, **166 Boulevard Macdonald Paris**), (destination, **59 Rue de la verrerie Paris**), (date, today), (time, 5 pm)*
                                         └──── *value inferred from history*   └──── *value inferred from history*

#4 User: Send a message to my friend, tell her "**I'll be at UGC cinema at 3 pm**"
  Bot: Done! I sent the message "**I'll be at UGC cinema at 3 pm**" to Hayet phone number **+33783425005**
INTENT: *SendMsg: (tel, ?) (msg, I'll be at UGC cinema at 3 pm)*
API-METHOD: *contacts-get: (contact, Hayet), (phoneNbr, +33783425005), (email, b.hayet@gmail.com)*
              *twilio-SendMessages: (tel, **+33783425005**) (msg, I'll be at UGC cinema at 3 pm)*
                                         └──── *value inferred by calling another API Method (i.e., contacts-get)*

#5 User: Just one last thing, can you start the **playlist** called **My Happy Melodies** on Spotify
  Bot: Sure, starting playlist ...Done!
INTENT: *StartPlaylist: (playlist-id, ?)*
API-METHOD: *Spotify-Search: (q, My Happy Melodies) (item, playlist) (spotify_id, 59ZbFPE)*
              *Spotify-Player: (spotify_id, **59ZbFPE**)*
                                         └──── *value inferred by calling the dependent method (i.e., Spotify-Search)*

#6 User: Oh, I also need a taxi to go from **21 Rue Maximilien Palaiseau Paris** to the Eiffel Tower **today** at **10 am**
  Bot: Done! I have booked a taxi from **21 Rue Maximilien Palaiseau Paris** to **Champ de Mars, 5 Av. Anatole France, Paris today** at **10 am**.
INTENT: *BookTaxi: (departure, 21 Rue Maximilien Palaiseau Paris), (destination, ?), (date, today), (time, 10 am)*
API-METHOD: *taxicode-booking: (pickup, 21 Rue Maximilien Palaiseau Paris), (destination, **Champ de Mars, 5 Av. Anatole France, Paris**),*
(date, today), (time, 10 am)   *value inferred from an external data service*

| Legend |
| --- |
| ■ Input parameter |
| ■ Output parameter |
| ■ Inferred value |

Fig. 1. Example of user-bot conversation where there are some complex intents.

Identify the dependent method to get a value of an id. The composite pattern called *API-calls ordering* allows mapping an intent to a sequence of API calls to satisfy order constraints. For example, in utterance #5, the user wants to start a playlist, but the value of the slot `playlist_id` is missing. In existing ST techniques, unless the bot developer implements an intermediate method that combines the two API methods `Spotify-Search` and `Spotify-Player`, the bot will ask the user for the value of the slot `playlist_id`. A bot improved with CKS will automatically map the user intent (i.e., StartPlaylist) to a sequence of API calls to get the value of the `id`.

Infer slot value from an external data service. The composite pattern called *entity-enrichment* allows resolving a missing value of a slot from an external data service. The user is not always precise; she might refer to an entity mention that is common knowledge to inform a slot value. For example, in utterance #6, the user provides "Eiffel Tower" as a taxi destination instead of the precise address. Since they leverage only the conversation history, existing ST techniques will fail to infer the value of the `destination` slot. A bot improved with CKS, however, can enrich the "Eiffel Tower" entity with additional information such as its address, which is the target destination value.

2.2   Architecture

To empower bots in handling the previous scenario, the bot needs services that initiate, monitor, and control conversations. Figure 2 shows the workflow be-
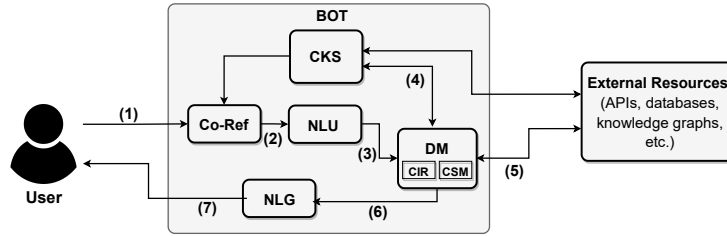
Fig. 2. General architecture supporting our approach.

tween these services. When a bot receives a new utterance, **(1)** the Co-Ref service takes this utterance and the previous messages as input and resolves the potential referenced mentions. We use Neuralcoref[4], as a coreference resolution model. Then, **(2)** the utterance is sent to the Natural Language Understanding (NLU) service to extract intent and slot values. We use DialogFlow[5] NLU model. The Dialogue Manager (DM) aims to coordinate the information flow in the conversation. In our approach, the DM uses a Conversational State Machine (CSM) model to represent bot behaviors [22] and relies on a Complex Intent Recognition (CIR) technique to identify complex intents (details in Section 4). **(3)** Once the DM gets the intent and slot values, it creates a composite state, in the CSM, if the CIR recognizes a complex intent. Otherwise it creates an ordinary state. **(4)** The CKS keeps track of conversation knowledge, infers missing slot values, and provides a set of new services supporting the recognition of complex intents (details in Section 3).

Once the DM collects all required information for the current state, **(5)** it calls the related API method and **(6)** sends the results to the Natural Language Generator (NLG). **(7)** NLG uses then predefined templates to generate human-like responses to the user.

3   Extended Context Knowledge Service

3.1   Context Knowledge Model

The context knowledge model consists of: (1) the metadata of bot schema, user profile and external services and (2) the data stored as conversation progresses.

Metadata is denoted as a context knowledge graph that includes mainly: (i) the definition of intents, slots, API methods, API parameters, and entity types, and (ii) relationships between these elements (e.g., `SearchCinema` intent is realized by the method `Yelp-SearchBusiness`, `departure` slot is same-as `address` parameter, etc.). Figure 3 illustrates an example of this graph related to the conversation scenario. The context knowledge graph includes also the metadata of external data services (e.g., Google KG, wikidata) and user profiles. The key intuition behind including external data services is to endow the bot with the capability of

---

[4]NeuralCoref: `https://spacy.io/universe/project/neuralcoref`
[5]DialogFlow: `https://dialogflow.com/`
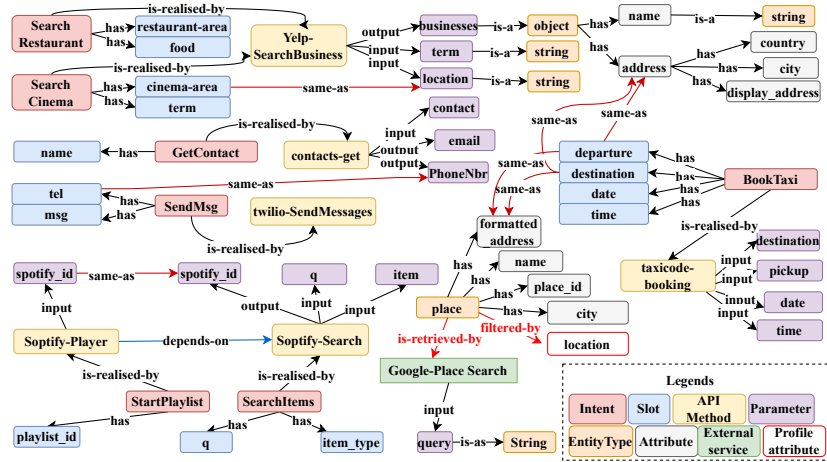
Fig. 3. Context knowledge graph related to the conversation of Figure 1. For clarity purpose we do not represent all nodes and edges.

enriching entities with additional information from these services. However, an external data service may return several entities for a given mention. Thus, having a mechanism that links the entity mention to its corresponding entity in the data service is necessary. This is where the user profile comes into play. User profile attributes like location, preferences can be used as filters to select the appropriate entity. Thus, we introduce the following node types: `External service`, `Profile attribute` and relationship types: `is-retrieved-by`, `filtered-by`. `External service` refers to a data service used for an entity enrichment. Often, a data service has an input parameter that takes a text query, which in our case will be filled by the entity mention (e.g., "Eiffel Tower"). `is-retrieved-by` is defined between an entity type and an external service, (e.g., the entity type `place` can be retrieved from `Google-PlaceSearch` service). `filtered-by` is defined between an entity type and a profile attribute, meaning that entities of that entity type can be filtered by the given attribute (e.g., `place` entities can be filtered by `location`). The generation of `filtered-by` edges is based on computing the cosine similarity [18] between the vector embedding[6] of each pair of entity type attributes and profile attributes. Assume that a user profile is defined by this set of attributes (`gender`, `location`, `dietary`) and the entity type `place` has the attribute `city` among others, a `filter-by` edge will be added between `place` and only `location` since it is similar to `city`. We assume that the external data services and their input parameters, the related `is-retrieved-by` relationships and the user profile attributes[7] are specified by the bot developer.

Data includes relevant information that should be memorized during user-bot conversations for later reuse. Two memory structures are used to store this data: Local Context Memory (LCM) and External Context Memory (ECM). The LCM

---

[6] Vector embeddings are obtained using the off-the-shelf spaCy NLP model

[7] Updating user profiles from conversations is out of the paper scope.

keeps track of all the traces related to each intent fulfillment. This includes the utterance, the intent, the method call, alongside with its timestamp, its inputs, and its outputs. The ECM, on the other hand, keeps track of all entities mentions in user utterances. It also provides all information that external data services extract to enrich these entities mentions. We structure ECM in terms of entities; each is associated with its mention, entity type, and its retrieved attribute values. Examples of data related to the scenario are documented in online appendix[8].

3.2   CK services

The CKS features four services, two of which are devoted to supporting the inference of slot values from the conversation history and external data services, whereas the two others aim at providing the contextual knowledge that is needed for the rules to recognize complex intents.

History search: This service allows inferring slot values from the conversation history. It is enabled by the endpoint "`CKS/history? ms & u`" which takes as inputs the missing slot `ms`, and the utterance `u` and returns the value of the slot `ms` when possible. First, the service extracts the relevant entity-mention pairs from the utterance. An entity-mention is relevant if the extracted entity has an attribute same-as the missing slot. Consider utterance #3,, the entity-mention (`restaurant, LaGoulette`) is relevant because the entity `restaurant` has an attribute `address` same-as the missing slot `taxi-destination`. Second, the service rewrites the utterance by replacing each mention with the corresponding attribute's value. For example, the utterance "book a taxi from UCG cinema to LaGoulette restaurant" will be "book a taxi from [UCG-address] to [LaGoulette-address]". The rewriting is important to know if the value of `taxi-destination` is the restaurant or the cinema address. The service then extracts the missing value from the new utterance. If there is no relevant entity-mention in the utterance, the service returns the most recent value of the parameters same-as the missing slot. For example, in utterance #2, there is no entity-mention, so the service returns the value of the restaurant's area.

Entity enrichment: This service allows enriching entity attributes from external data services. It is enabled by the endpoint "`CKS/invoke_external_service? s & em & a`". Consider utterance #6, the service takes as inputs: the external data service `s: Google-PlaceSearch`, the entity-mention `em: (place, Eiffel Tower)`, and the attribute `a: address` and it returns the value of `a`. To obtain the attribute value from the appropriate entity, three steps are followed. First, the service invokes the external data service related to the given entity-mention `em`, which returns a set of entities. Then, it filters the returned entities by discarding any entity, whose similarity with the entity-mention `em` is less than a predefined threshold and it does not contain the target attribute value. The similarity is computed on the basis of the cosine distance between the embedding vectors of the entity-mention `em` and the name of the entity returned by the external

---

service. After this step, if only one entity is returned, the service retrieves the target attribute value from it. Otherwise, in order to identify the right entity, the service proceeds a second filtering step based on the filter attributes related to the mention entity type in the metadata. This filter step is expected to return one entity that matches the most of filters while giving a high priority to the location filter.

Nested method identification: This service allows identifying an API method that needs to be invoked to obtain the missing value. It is enabled by the endpoint "`CKS/nested_method? ms & set_em`". Consider utterance #4. The service takes as inputs the missing slot `ms: tel`, and the set of detected entity-mentions `set_em: {(person, Hayet)}`. It then gets from the metadata the methods that have an output parameter same-as the slot. For example, the service gets the set of methods `{contacts-get, businessDetails-get}` where the missing slot `tel` is the same-as one of the outputs of `contacts-get` (i.e., `phoneNbr`) and also the same-as one of the outputs of `businessDetails-get` (i.e., `phone`). The service relies on the detected entity-mentions to select the relevant method from the set of methods. For example, in contrast to the method `businessDetails-get`, the method `contacts-get` has an input parameter `contact` same-as to one of the detected entity `person`. Thus, the service selects `contacts-get` as the nested method and returns it along with its input values `{(contact, Hayet)}` and one of its outputs `phoneNbr` that is same-as the slot `tel`.

Dependent Method identification: This service allows identifying a dependent method to get a value of an id. It is enabled by the endpoint "`CKS/dependent _method? i`" where `i` is the given intent. This endpoint first gets the method `m1` that realize the intent `i`. Then, it gets the dependent method `m2` where `m1` depends on `m2`. For example, in utterance #5, the endpoint takes the intent `StartPlaylist` as input and returns: the dependent method `Spotify-Search`, its input parameters `{q, item}`, and its `id` output parameter `spotify_id`.

4 Complex Intent Recognition

We propose a new rule-based approach to support Complex Intent Recognition (CIR). Our approach offers flexibility for an extensible library of dialog patterns, i.e., when a new class of a complex intent is identified, we can add its new rule to recognize the intents of this class from user utterances. We express a rule using a combination of natural language features and contextual knowledge. In what follows, we first define functions that we use to specify the rules, then we specify the rule of each pattern introduced in Section 2.1.

4.1 Functions

Functions are the primitives that we use to define the rules. We consider function input and output types to be standard data types found in common programming languages such as `string` and `boolean`; as well complex data types such as `Tuple` or `Set`. Thus, we can leverage the standard operators designed for these data

Table 1. Examples of boolean functions to express triggers

Functions	Inputs	Description
`IS_NEW_INTENT()`	u: string	returns `true` if the identified intent in the utterance u is a new intent.
`HAS_MISSING_SLOT()`	u: string s: string	returns `true` if the value of the given slot s is not recognized in the utterance u.
`HAS_SAMEAS_PARA()`	s: string	returns `true` if there is at least one parameter that is the `same-as` the slot s.
`EXIST_NESTED()`	s: string	returns `true` if there is at least one output parameter that is the `same-as` the slot s.
`IS_DEPENDENT()`	i: string	returns `true` if the method that realize i `depends on` another method.
`HAS_SAMEAS_ATT()`	set_em: set s: string	returns `true` if at least one entity in `set_em` has an attribute that is the `same-as` the slot s.

types. We distinguish two types of functions: dialog act functions to capture natural language features and context metadata functions to capture contextual knowledge. These functions are offered by the NLU and the CKS, respectively.

Dialog Act functions identify hidden actions in user or bot messages. Whether the user is providing information, or asking a question, or the bot is providing suggestions, are all hidden acts in user or bot messages. We focus on two dialog act functions: `INTENT_OF()`, which identifies the intent `i` expressed in a given utterance `u`, and `SLOT_VALUE()`, which returns the value of a given slot `s` recognized in the utterance `u` or `NULL` if no value of `s` is recognized in `u`.

Context Metadata functions allow to access and query the metadata graph defined in Section 3.1 to get the contextual knowledge. For example: `GET_SAMEAS_PARA()` is a metadata function that returns a set of parameters that are the same-as a given slot `s`; `DEPENDS_ON()` returns a method name m_b given a method name m_a where m_a depends on m_b or it returns `NULL` if there is no dependent method.

4.2 Complex Intent Recognition Rules

A rule consists of *trigger* and *action* clauses. The trigger clause specifies the conditions that need to be verified to recognize complex intents. Then, the sequence of operations specified in the action clause are executed to fulfill the related complex intent. The following statement specify a rule:

> `Rule` "name of the rule" **when trigger then action**

Triggers are expressed as boolean conditions over functions, including dialog act and metadata functions. Table 1 provides examples of boolean functions that are used to define triggers. Conditions may be combined using conjunction operator (`AND`). The action is a sequence of operations. For instance, an operation can be an assignment of a value to a given variable, or an invocation of a CKS service. Figure 4 shows the definition of rules for each composite pattern:

Slot-value-flow Rule. The first condition checks if the identified intent `i` is a new intent. The second condition checks if the value of the slot `ms` is missing. These two conditions are the same for nested-method and entity-enrichment rules. The third condition checks if there is at least one already fulfilled parameter that is the `same-as` the slot `ms`. If the conditions are satisfied, the bot: **(1)** invokes

```
Rule "slot-value-flow"
when
IS_NEW_INTENT(i).equals(true)    AND
HAS_MISSING_SLOT(u, ms).equals(true)      AND
HAS_SAMEAS_PARA(ms).equals(true)
then
mv := "CKS/history? ms & u"
set_sv_i := set_sv_i ∪ {(ms, mv)}
INVOKE(i, set_sv_i)
```

```
Rule "nested-method"
when
IS_NEW_INTENT(i).equals(true)    AND
HAS_MISSING_SLOT(u, ms).equals(true)      AND
EXIST_NESTED(ms).equals(true)
then
(m_nes, set_iv_nes, o_nes) := "CKS/nested_method? ms & set_em"
mv := GET_OUTPUT_VALUE(m_nes, set_iv_nes, o_nes)
set_sv_i := set_sv_i ∪ {(ms, mv)}
INVOKE(i, set_sv_i)
```

```
Rule "entity-enrichment"
when
IS_NEW_INTENT(i).equals(true)    AND
HAS_MISSING_SLOT(u, ms).equals(true)      AND
HAS_SAMEAS_ ATT(set_em, ms).equals(true)
then
(s, em, a) := GET_REQUIREMENTS(set_em, ms)
mv := "CKS/invoke_external_service? s & em & a"
set_sv_i := set_sv_i ∪ {(ms, mv)}
INVOKE(i, set_sv_i)
```

```
Rule "API-calls-ordering"
when
IS_NEW_INTENT(i).equals(true)    AND
IS_DEPENDENT(i).equals(true)
then
(m_dep, set_i_dep, id_dep) := "CKS/dependent _method? i"
set_iv_dep := GET_VALUES_ASKUSER(m_dep, set_i_dep)
idv := GET_OUTPUT_VALUE(m_dep, set_iv_dep, id_dep)
set_sv_i := set_sv_i ∪ {(id, idv)}
INVOKE(i, set_sv_i)
```

Fig. 4. Rules of composite dialog patterns

the history search CKS service to get the missing value, **(2)** adds this value to the set of slot-value pairs, and **(3)** invokes the method that realize the intent `i`.

Nested-method Rule. The third condition checks if there is at least one output parameter that is the `same-as` the slot `ms`. If the conditions are satisfied, the bot: **(1)** invokes the nested method CKS service to identify: the nested method m_{nes}, its input values set_iv_{nes}, and its output parameter o_{nes}. Then, the bot **(2)** invokes the method m_{nes} to get the value of o_{nes}, **(3)** uses this value as a value for the slot `ms`, and **(4)** invokes the method that realize the intent `i`.

API-calls-ordering Rule. The first condition is similar to the first condition of the other rules. The second condition checks if the method that realize the intent `i` depends on another method. If the conditions are satisfied, the bot: **(1)** invokes the CKS service to identify: the dependent method m_{dep}, its inputs set_i_{dep}, and the id parameter id_{dep}. Then, the bot **(2)** calls `GET_VALUES_ASKUSER()` to get the input values of the method m_{dep} by extracting them from the utterance, the history, or by asking the user. After getting the input values, **(3)** invokes m_{dep} to get the value of id_{dep}, **(4)** uses this value as a value for the parameter `id`, and **(5)** invokes the method that realize the intent `i`.

Entity-enrichment Rule. Given a set of entities mentions `set_em`, extracted from `u`, the third condition checks if there is at least one attribute of an entity-mention that is the `same-as` the slot `ms`. If the conditions are satisfied, the bot: **(1)** calls the metadata function `GET_REQUIREMENTS()` to get the following information: the related service `s`, the entity-mention `em`, and the attribute `a`. Note that this function chooses one entity-mention from `set_em` based on the one that has an attribute `same-as` the slot `ms`, **(2)** invokes the entity enrichment CKS service to get the value of the attribute, **(3)** uses this value as a value for the slot `ms`, and **(4)** invokes the method that realize the intent `i`.

5   Experiments

The first objective of the study was to explore the *effectiveness* and limitations of (i) the proposed CKS (i.e., its capability of inferring slots' values correctly and reducing unnecessary interactions) and (ii) the CIR (i.e., its capability of recognizing correctly the composite dialog patterns mentioned previously). The second objective was to evaluate the user experience (i.e., *naturalness*, *repetitiveness*, *understanding*) in interacting with a bot improved with CIR and CKS.

5.1   Methods

Experimental design. Participants were recruited via email from the extended network of contacts of the authors. The call for volunteers resulted in a total of 20 participants. We prepared an evaluation scenario that required participants to interact with a set of API methods through a bot to plan an evening activity in Paris. Participants were asked to complete four different tasks in this scenario (T1: checking the weather and searching for restaurants, T2: booking a restaurant table, T3: booking a taxi, and T4: sending a confirmation message to the travel partner). The tasks were designed to leverage the type of support provided by the CKS, if the composite dialog patterns were to be effectively recognized (T1: inferring slot value from conversation history, T2: identifying dependent method, T3: using an external data source, and T4: identifying nested method). We followed a within-subjects design,[9] tasking participants to interact with two bots representing the following experimental conditions:

  – *DM-Baseline*. The baseline implements a standard conversational management, without composite dialog patterns and CKS support.
  – *DM-CKS*. This bot is implemented with the CIR and CKS support.

    The two bots relied on the same NLU implementation (in DialogFlow), bot interface, and differed only in the composite patterns and CKS support.

Procedure. The study was conducted online. Participants received a link to an online form that included an informed consent, all the instructions, links to the bots and feedback required. In the study, participants were introduced to the evaluation scenario and tasks, and were asked to perform those tasks with the two bots. The order in which the bots were presented to users was counterbalanced. For each bot, participants were asked to provide open-ended feedback on the pros and cons of their experience. The last part of the study then asked participants about their preferred bot, the reason why, and a quantitative feedback on their user experience. We adopted the user experience questions from prior work [9], to get feedback on the perceived *naturalness* (i.e., ability to fulfill user tasks in human-like conversations), *repetitiveness* (i.e., ability to avoid redundant questions) and *understanding* (i.e., ability to interpret user requests).

Data analysis. We performed an analysis of conversation logs so as to assess the effectiveness of the CKS and the CIR. These are calculated in relation to optimal

---

[9]Study materials and in-depth results available at `https://tinyurl.com/study-materials`

conversation scenarios[10] that we designed based on participants conversations. The CKS effectiveness is calculated by considering the following metrics: number of (M1) conversation turns, (M2) prompts asking for missing slot values, and (M3) missing slot values correctly inferred. The effectiveness of the CIR (only available in DM-CKS) is calculated by considering the number of (M4) complex intents correctly detected. These metrics are calculated per user conversation, aggregated (mean) and then used to compute the relative performance against the optimal scenario. We also performed a qualitative analysis of open-ended responses and conversation logs to contextualise the results from the metrics and identify limitations.

5.2 Results

Effectiveness of CKS and CIR. Table 2 shows the relative performance by task of both bots DM-Baseline and DM-CKS in relation to the optimal reference scenario. For the four tasks, we can see that DM-CKS bot experienced a boost in performance for M1 and M2 metrics (mean across tasks 94.66% and 88.4% respectively), approaching the efficiency in terms of number of turns and prompts of the reference ideal scenario. This level of performance is possible due to the accuracy of the slot value inference (M3) performed by the CKS services supporting each task – a mean relative performance across tasks of 96%. In contrast, not having the support of the CKS services lead the DM-Baseline bot to perform poorly in comparison, with the best performance being at around 37.18% for the metrics considered. These results provide evidence for the benefits and effectiveness of the CKS support. Table 2 also shows the relative performance of recognizing complex intent (M4) by the bot DM-CKS in relation to the reference scenario. By analyzing the conversations, we noticed that the recognition error of the composite dialog patterns is mostly caused by the detection error of the correct intent by the NLU during the conversation. For example, if the NLU detects the intent `SearchRestaurant` instead of `BookTaxi`, in the utterance "I want to go to this restaurant", this will lead to an error in detecting the slot-value-flow pattern that takes the restaurant address as the destination address. However, for the four tasks, we can see that DM-CKS bot is close to the reference scenario with a mean relative performance across tasks of 94.86%.

User experience. All but one participant (19/20 participants) expressed a preference towards the DM-CKS bot as opposed to the baseline. The one exception was due to NLU limitations in recognizing user expressions that led to the enactment of the wrong services. The feedback to the specific user experience questions, as well as the open-ended feedback, highlighted the reasons behind the preference. Participants agreed with DM-CKS interactions describing *naturalness* (14/20), less *repetitiveness* (16/20) and *understanding* (15/20), whereas the baseline was poorly rated on these fronts (1/20). Interestingly, these qualities were linked to the CKS support, such as the ability to infer missing slot values from conversation history (e.g., "saying that the drop-off address was the

---

[10]Scenarios assuming ideal accuracy of slot-value inference and intent recognition.

Table 2. Bot performance for each task according to relevant metrics. Values in bold denote best performance. Percentages denote the relative performance with respect to the reference (optimal) scenario.

Task (service)	DM-Baseline			DM-CKS			
	M1 (TURNS)	M2 (PROMPTS)	M3 (SLOTS)	M1 (TURNS)	M2 (PROMPTS)	M3 (SLOTS)	M4 (PATTERN)
T1 (history)	59.70%	21.18%	49.24%	**98.04%**	**90%**	**96.97%**	**95%**
T2 (dependent)	61.18%	42.11%	17.86%	**92.86%**	**95.24%**	**95.24%**	**94.44%**
T3 (external)	46.97%	32.39%	3.17%	**91.18%**	**88.37%**	**95.24%**	**95%**
T4 (nested)	52.97%	20%	39.41%	**96.55%**	**80%**	**96.55%**	**95%**
Mean	55.21%	28.92%	27.42%	**94.66%**	**88.4%**	**96%**	**94.86%**

restaurant I have just booked was enough", P1), or from external services (e.g., "[it] found the address when I said Eiffel Tower", P6). The ability to handle complex intents also emerged as a defining feature (e.g., "[DM-CKS] is capable of undertaking complex tasks and retaining previous information", P16). In contrast, participants reported having to copy & paste previous values or google some information during their interactions with the baseline bot.

Limitations. The conversation analysis revealed some limitations in supporting the natural language interaction described by the users:

Enumerating entities when the number of entities is expected. In the context of T2, when asked "For how many people do you want to book a table?", some participants would respond with "For <u>me</u> and <u>my friend</u>". The bot could not infer the number of seats from the participant's utterance because it expected to extract a number. This would be an acceptable answer in a natural conversation, and represent as a new type of inference that needs to be considered.

Introducing typos when providing slot values. When booking a taxi in the context of T3, some participants spelled "Eiffel tower" incorrectly (e.g., "Book a Taxi from the Eifeltower"), which led to the failure of the CKS service for inferring values from a external data source. Handling mistakes when performing inferences is a situation that needs to be addressed.

6   Related Work

Our work is related to the ST process that aims to infer the dialog state in terms of the user intent and its slot-value pairs during conversations [15]. Depending on the leveraged knowledge sources, existing ST approaches can be organized into history-based, schema-based, and Linguistic patterns (LPs) based.

History-based approaches rely on the whole or window-size of the dialogue history to predict the dialog state. Deep learning models including HRNN [8], LSTM [7] and BERT [23] are utilised to encode the dialog history. Other works [4,17] leverage only on the previous dialog states to predict the current state instead of taking the whole history. More advanced approaches [14,20,10] focused first on the learning of slot dependencies from the history and then incorporated them into the ST model, allowing it to infer slot values from similar slots. Most of the ST approaches either focused on recognizing only basic intents or ignored their recognition at all. Similar to some of these, we build upon advances in

ML techniques to enable the recognition of basic intents but contribute a new rule-based approach to recognize complex intents.

Schemas-based approaches leverage schemas capturing the structural representation of conversation data to predict the dialog state. Works like [5], [13] [19] use a slot-level schema graph that captures dependencies between slots. The aim is to allow the ST model to infer slot values from similar slots. Other efforts like [12] leverage the backend database schema in ST model to allow slot inference from the database entities. These methods, however, work with bots integrated only with databases, where the inferred slot-value pairs are used to frame the query. Since our context knowledge model integrates API/Service schema, where each intent is associated with its corresponding API method, our approach can handle flows supported by software-enabled services. The work [1] represents the dialogue state as a dataflow graph and the complex user intent as a dataflow program. For each user utterance, a trained model allows predicting the corresponding dataflow program. This approach relies on datasets where each utterance must be annotated with the corresponding dataflow program; however, it is not intuitive task to annotate utterances with programs. The closest work to ours is [15] which introduced a unified schema defining a service or API as a combination of intents and slots. A BERT-based ST model then takes this schema as input to enable the recognition of intents and inferring their slot-value pairs. The captured knowledge (i.e., the unified schema), however, can only help in the recognition of basic intents. In our work, we devise the CKS mainly to capture the contextual knowledge that is required to recognize complex intents. In addition to the conversation history and intent/slot schemas, this knowledge includes API/Service schemas and enriched entities. Our CIR technique exploits both this knowledge and basic intent features to recognize the complex intents.

LPs-based approaches leverages linguistic patterns that are drawn mainly from human conversation to handle some complex intents and inferring their slots [6], [16]. For example, IRIS [6] draws on two existing LPs *dependent questions* (i.e., one question depends on the answer to some subsequent request), and *anaphora* (i.e., expressions that depend on previous expressions) to allow composition and sequencing of intents.These approaches, however, are not enough to capture complex intents that naturally emerge when conversing with services. For instance, while this utterance "Send the message 'I will be at UGC cinema at 3 pm' to Hayet" refers to a complex intent requiring a composition of two API methods, it cannot be recognized by IRIS. The reason is that IRIS can recognize composition based only on linguistic features (e.g., A composition is recognized when a user answers with a new intent to the bot request for a missing slot). In contrast to the LPs-based approaches, we focus on using composite dialog patterns that cater to the inherent features in interactions between humans, bots, and services in addition to the linguistic ones. These patterns are used to enable the contextual knowledge required by the CIR to recognize complex intents.

7   Conclusions and Future Work

We proposed reusable and extensible rule-based technique that uses a sophisticated context service to recognize and realize complex intents. We believe that our approach charts novel abstractions that unlock the seamless and scalable integration of natural language-based conversations with software-enabled services. We devised a novel complex intent recognition that allows the incremental acquisition of rule templates to identify composite intents from basic dialog acts and context features. The contextual knowledge required at run-time to recognise complex intents and infer slot values from user-bot conversations is extracted from conversation history, enriched entities, intents and API schemas and represented in graph structure. Future work includes identifying new composite patterns (e.g., supporting conversations with business process models) and developing privacy-aware task-oriented bots by reasoning about privacy-preserving conversations.

References

1. Andreas, J., et al.: Task-oriented dialogue as dataflow synthesis. Transactions of the Association for Computational Linguistics **8**, 556–571 (2020)
2. Bouguelia, S., et al.: Reusable abstractions and patterns for recognising compositional conversational flows. Proc. CAiSE 2021
3. Brabra, H., et al.: Dialogue management in conversational systems: A review of approaches, challenges, and opportunities. IEEE TCDS (2021)
4. Chao, G.L., Lane, I.: Bert-dst: Scalable end-to-end dialogue state tracking with bidirectional encoder representations from transformer. pp. 1468–1472 (09 2019)
5. Chen, L., et al.: Schema-guided multi-domain dialogue state tracking with graph attention neural networks. Proc. AAAI Conference on Artificial Intelligence (2020)
6. Fast, E., et al.: Iris: A conversational agent for complex tasks. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (2018)
7. Gao, S., Sethi, A., Agarwal, S., Chung, T., Hakkani-Tür, D.Z.: Dialog state tracking: A neural reading comprehension approach. In: SIGdial (2019)
8. Goel, R., Paul, S., Hakkani-Tür, D.: Hyst: A hybrid approach for flexible and accurate dialogue state tracking. In: Interspeech (2019)
9. Holmes, S., et al.: Usability testing of a healthcare chatbot: Can we use conventional methods to assess conversational user interfaces? In: Proc. of ECCE (2019)
10. Hu, J., et al.: SAS: Dialogue state tracking via slot attention and slot information sharing. In: ACL. Association for Computational Linguistics (2020)
11. Jain, M., Kumar, P., Kota, R., Patel, S.N.: Evaluating and informing the design of chatbots. In: Proc. of the 2018 Designing Interactive Systems Conference (2018)
12. Liao, L., Long, L.H., Ma, Y., Lei, W., Chua, T.S.: Dialogue State Tracking with Incremental Reasoning. TACL **9**, 557–569 (2021)
13. Lin, W., Tseng, B., Byrne, B.: Knowledge-aware graph-enhanced GPT-2 for dialogue state tracking. CoRR (2021)
14. Ouyang, Y., et al.: Dialogue state tracking with explicit slot connection modeling. In: ACL (2020)
15. Rastogi, A., et al.: Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. ArXiv **abs/1909.05855** (2020)

16. Rastogi, P., Gupta, A., Chen, T., Lambert, M.: Scaling multi-domain dialogue state tracking via query reformulation. In: NAACL. pp. 97–105 (2019)
17. Ren, L., Ni, J., McAuley, J.: Scalable and accurate dialogue state tracking via hierarchical sequence generation. In: EMNLP (2019)
18. Sitikhu, P., et al.: A comparison of semantic similarity methods for maximum human interpretability. In: Proc. AITB (2019)
19. Wu, P., et al.: Gcdst: A graph-based and copy-augmented multi-domain dialogue state tracking. In: Findings of the ACL: EMNLP 2020 (2020)
20. Ye, F., et al.: Slot self-attentive dialogue state tracking. In: WWW. ACL (2021)
21. Zamanirad, S.: Superimposition of natural language conversations over software enabled services (2019)
22. Zamanirad, S., et al.: Hierarchical state machine based conversation model and services. Proc. CAiSE 2020
23. Zhang, J., et al.: Find or classify? dual strategy for slot-value predictions on multi-domain dialog state tracking. In: STARSEM (2020)