The Workshops of the Thirty-Second
AAAI Conference on Artificial Intelligence

Knowledge-Graph Driven Information State Approach to Dialog

Svetlana Stoyanchev, Michael Johnston
Interactions Corporation
25 Broadway, New York, NY

Abstract

A modular conversational dialog system, in contrast to end-to-end, includes natural language understanding, dialog management, and natural language generation components. A dialog system framework simplifies development and maintenance of modular dialog systems. We propose a knowledge-graph driven framework (KGD) based on the Information State Update (ISU) approach and adapted for practical task-oriented applications. With the proposed framework, a system is defined declaratively by describing the information structure of a domain. We demonstrate the effectiveness of the approach in enabling rich conversational dialog in food ordering domain.

Introduction

We propose a knowledge-graph driven dialog management framework (KGD) derived from the Information State Update (ISU) approach (Traum and Larsson 2003). A key property of an information state-based theory is that the dialog "information" is encoded in the state itself. Unlike approaches commonly used in many commercial dialog systems, where possible dialog patterns are explicitly laid out by a dialog designer as a series of pre-determined dialog states or nodes with fixed transitions among them, in an ISU system the set of dialog states and possible transitions among states are not authored directly, rather the dialog flow emerges from the application of a series of rules operating over a structured data representation capturing the current dialog state at each point in the interaction. This allows for a more compact implementation of mixed initiative dialog functionality. For example, in an information gathering task (e.g. capturing a hotel reservation), a user may specify pieces of required information (e.g. dates, number of guests, property) all together in a single turn or in multiple turns and in any order, not necessarily directly responding to system prompts. An ISU-based system defined by a small set of rules can handle mixed initiative dialogs of this type, however implementing this functionality with a finite-state approach would involve significantly more complexity. To encode mixed initiative with a finite-state approach, we would need a state for each combination of inputs and transitions between all of them.

Our motivation is to provide a framework enabling rapid authoring and easy maintenance of robust and flexible mixed initiative dialog systems that can be deployed commercially. The ISU formalism has the flexibility to support these needs. It also allows for potential incorporation of research components based on machine learning and reasoning into ISU-based systems. We want this flexibility in commercially deployed systems to evaluate research questions with real users narrowing the gap between research and commercial dialog systems without compromising the user experience. However, authoring, debugging, and maintaining an ISU system requires specialized AI expertise. We propose several modifications to the ISU approach which, we believe, will simplify system development and maintenance and improve dialog system's robustness to errors and ambiguities in the natural language input.

First, we change the dialog system authoring method. An ISU system is defined as a set of rules with *pre-* and *post-*conditions. A dialog engine selects and executes the rules based on the match between the state and the pre-condition. In contrast, a KGD-based dialog system is authored declaratively by specifying its domain-specific information structure. A domain information structure of an application, such as airline reservation, banking, or food ordering may be defined by a domain expert. It corresponds to the type of information that can be exchanged between a user and a system. For example, in the food ordering domain, information structure corresponds to a restaurant menu, augmented with constraints regarding special combinations or promotions. This domain definition initializes the system with capabilities to add, remove, or modify menu items.

Second, we change the representation of *move*. In the ISU approach, *move* corresponds to a dialog act, such as **ask** and **answer**. We shift the meaning of the *move* away from language and closer to the domain functions. A *move* in the KGD approach defines an unambiguous single symbolic modification of the information state. At each turn, a set of possible user moves is generated based on the current information state and (static) domain knowledge graph. A *move* is context specific: *'add peppers to the 0'th pizza item'* will only be generated if there is at least one pizza item in the information state.

Third, we change the method of *move* selection. In the ISU approach, being able to execute a correct set of moves

relies on a match between an utterance interpretation and a rules's pre-conditions. This potentially suffers from error propagation: if utterance interpretation result is incorrect, the error will be propagated to the *move* selection. In the KGD approach, we score the automatically generated possible moves with a scoring function which can be rule-based or learned from data. It can take into account N-best hypotheses from the interpretation module, confidence scores, domain-specific prior probabilities, etc. The move scoring serves as a contextual disambiguation making the dialog manager more resilient to errors in recognition and interpretation components.

We will demonstrate a capability of creating and modifying declarative domain representation and deploying the system in real-time on an AWS server.

System Architecture

1	U:	what do i have so far
2	S:	Your order is empty. What can I get for you?
3	U:	a large pizza with pepperoni
4	S:	Added a large pepperoni pizza, Anything else dear?
5	U:	i also want a pizza with mushrooms and onions
6	S:	Added a mushroom and onion pizza. What size do you want?
7	U:	show the menu
8	S:	[MENU] What is the size for the mushroom and onion pizza?
9	U:	small
10	S:	OK, I set the size of mushroom and onion pizza to small. What else can I get for you?
11	U:	remove onions from the small
12	S:	Removed onions from the small mushroom and onion pizza. Anything else dear?
13	U:	what do i have so far
14	S:	You ordered 2 Pizzas: large pepperoni pizza, and a small mushroom pizza. What else can I get for you?

Table 1: Example Dialog with the *PizzaOrdering* KGD system.

Figure 1 illustrates KGD system architecture. A domain-specific NLU component labels intents and entities in a user utterance. NLG generates system utterance from a template and includes a domain-specific code for generating referring expressions. All other system components are generic.

We implement the proposed framework with the support for the data collection functionality. The framework functionality may be extended by adding new node types to support other agendas such as query and response navigation.

Next, we describe KGD domain and state representations, and the generic system components.

Domain Representation

A dialog system author defines the domain information structure in a graph (represented in json) that is used to initialize a domain-independent dialog manager. In food ordering, a KGD-domain description can be derived from a restaurant menu. Table 1 shows an example dialog between a user and KGD dialog system initialized with the *PizzaOrdering* domain definition (Figure 2).

Each node in the KGD language is either a *decision*, a *form*, or a *leaf* node. A set of type-specific properties, including NLU and NLG templates, is associated with each node. During move selection, the intent and entity types assigned by the NLU component are matched with the NLU templates from the domain representation. A root *decision* node in Figure 1 is ORDER. Its $NLG\_request$ property defines templates for a generic system request *"Anything else, dear?", "What else can I get for you?"*.

Form nodes (e.g. PIZZA and DRINK) are the lowest non-leaf nodes representing domain information that can be exchanged between a user and a system and stored in the information state. A *list* is a special case of a *form*. Multiple instances of a *list* node type may be added to the information state. In food ordering, *list* nodes corresponds to the menu items. List-specific properties are NLU and NLG templates for adding and removing an item.

A *leaf* node is a child of a form node. Each leaf node corresponds to an individual property of an item (e.g. TOPPING, SIZE, or CRUST). A leaf node is associated with an entity type NLU_entity that matches entity label assigned by the NLU to the user text. The $type : multi$ in the $TOPPING$ node indicates that the node can take multiple values. NLU_set / NLU_rm and NLG_set / NLG_rm specify intents and response templates for setting or removing a property of the TOPPING node. The user utterance on line 11 triggers an instantiation of the NLG_rm template of the TOPPING node on line 12 where _VAL is resolved to *'onions'* and _REF, to *'small mushroom and onion pizza'*.

State Representation

Information state in a dialog system corresponds to the common ground, the information exchanged between a system and a user. To represent information state, we follow the ISU approach modifying the dialog game board initially proposed by (Ginzburg 1996). KGD information state stores a set of shared beliefs and dialog history, including last move and question under discussion. We assume that the system agenda is static and derived from the domain knowledge graph. The shared beliefs are stored as *json* object corresponding to the *form* nodes in the knowledge graph. In a *PizzaOrdering* domain, an information state represents the current order (see S_0 and S_1 in Figure 3). A *move* in the KGD approach defines an unambiguous single symbolic modification of the information state represented with *json* structure. A *move* may be adding a new pizza item using one of the attributes or adding/removing a topping on an existing pizza item, or changing a pizza size. We implement an 'execute-move' operator that takes as input a state S_0 and a set of moves $\{M\}$ and generates a new state S_1. Figure 3 shows an example move execution: S_0 with two pizza items (one large pepperoni and mushrooms and one pepperoni) is modified with two dialog moves $\{M\}$ (adding onions and peppers topping to the first pizza item) and the resulting state S_1.

Dialog Manager

KGD dialog manager consists of the domain independent components: $MoveGenerator$, $Encoder$, $MoveScorer$,

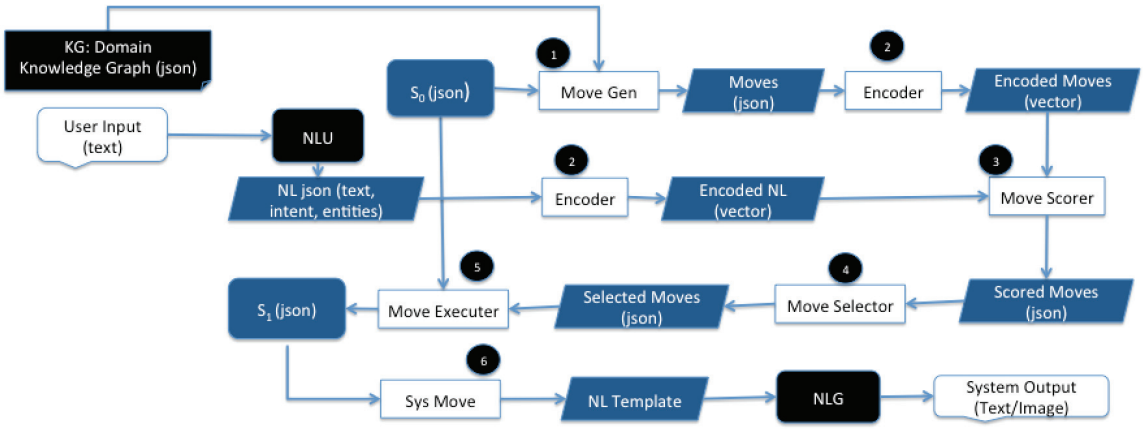

Figure 1: System Architecture

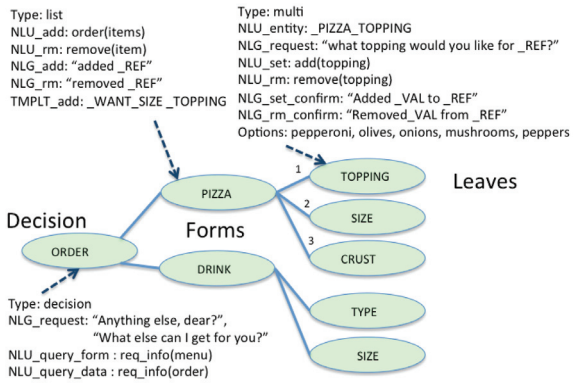

Figure 2: Pizza ordering domain

$Selector$, $Executor$, and $SystemMove$. Figure 1 outlines the components and processing sequence:

1. Move Generator (MG) is initialized with the static domain knowledge graph and current information state (S_0). MG deterministically generates a set of possible user moves. The algorithm enumerates all legal modifications of the current state based on the domain definition. For example, in a pizza ordering domain, when an information state (corresponding to an order) contains two pizza items, the set of possible moves includes: adding a new item from the menu, removing or modifying one of the pizza item. The types of possible modifications are based on the domain description of a particular item (e.g. add/remove a topping, change size). Each generated move has a *_sem* and an *_act* parts. *_sem* represents semantics of a user utterance (e.g. $\{Add(onions), mention(SIZE : large, TOPPING : pepperoni, mushroom)\}$. The *_act* part encodes the action to be performed on the information state if this move is selected.

2. Encoder (ENC) converts the *_sem* part of a move and a user's NLU-processed input into a vector representation. We implement an NLU-based encoder that uses domain-specific intents, labels, and values as vector dimensions.

3. Move Scorer (MSC) assigns a score to each move by computing the likelihood that a move matches utterance semantics. In our experiments, we use dot product to compute the score of each potential move. Move scorer estimates a probability of a match between an utterance and a move. Moves that have closer semantics to the utterance NL will receive higher score. If NLU output contains errors, a correct move may still get the highest score based on the context.

4. Move Selector (MSEL) identifies a set of moves to be executed based on their scores. Since a user utterance may contain multiple moves (e.g. add multiple items, or modify multiple toppings), multiple top moves may be selected. We implement a TOP-ITEM heuristics: select all moves that correspond to the same item.

5. Move Executor (EX) executes the *_act* part of the selected moves and generates a new state illustrated on Figure 3.

6. System Move (SYS) ensures that there is an open question in the system's turn. SYS finds an 'incomplete' node in the information state and adds its $NLG\_request$ to the agenda. On line 5, a user adds a pizza item by specifying two toppings. Because the size of this item is unspecified, the system requests the size for this item on lines 6 and 8 in Table 1. If all items in the information state are 'complete', SYS backs off to a $NLG\_request$ of the root ORDER node (lines 2, 4, 10, and 12).

Evaluation

Using KGD domain specification language, we define two dialog systems in *FoodOrdering* domain: pizza and burger ordering. The NLU component is trained on synthetic data generated from expert-authored templates[1]. The application-specific part of the NLG component consists of 50 lines of a

[1]We reuse the NLU model trained for a pizza ordering demo system developed without the KGD framework.

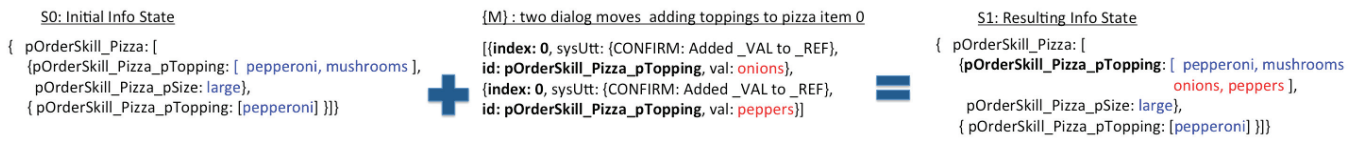

Figure 3: Move execution operation $S_0 + \{M\} = S_1$

python code to resolve _REF template into a reference for a menu item.

Ten colleagues not involved in the project interacted with the pizza ordering web chat interface. The users follow a loosely defined script by adding, removing, and modifying menu items using natural language sentences. The users self-reported errors in system responses. We manually analyzed the system action marked as incorrect by the uses. On 169 user utterances, we observed 7% of error where some of the errors were caused by an error in the NLU component or a user speaking out of domain.

Conclusions and Future Work

In this paper, we described a framework for authoring knowledge-graph driven dialog systems. KGD extends the idea of system generation from a form (Stoyanchev, Lison, and Bangalore 2016) to a more general task of system generation from a knowledge graph. Our approach is motivated by the previous dialog management frameworks that simplify task-oriented dialog system authoring and facilitate reuse of generic components across domains (Allen et al. 2000; Xu and Rudnicky 2000; Bos et al. 2003; Bohus and Rudnicky 2003; Lison and Kennington 2016). KGD diverges from these frameworks in the method of system authoring: by describing domain information structure in a knowledge graph, drawing motivation from ontology-based systems (Sonntag et al. 2009; Wessel et al. 2017).

We demonstrate the effectiveness on the proposed approach on a food ordering data collection task. With the KGD framework and using mostly generic code base, a fully functional dialog manager is created declaratively in minutes. In the future work, we will extend the KBD framework to support other types of agenda, including query and result navigation.

Hybrid methods that combine complementary strengths of knowledge-driven and statistical approaches (Mittal, Joshi, and Finin 2017; Williams, Asadi, and Zweig 2017) requre significantly less training data in comparison with the pure end-to-end methods (Bordes and Weston 2016). We will develop a hybrid KGD with a data driven move scoring and move selection components.

References

Allen, J.; Byron, D.; Dzikovska, M.; Ferguson, G.; Galescu, L.; and Stent, A. 2000. An architecture for a generic dialogue shell. *Nat. Lang. Eng.* 6(3-4):213–228.

Bohus, D., and Rudnicky, A. 2003. Ravenclaw: Dialog management using hierarchical task decomposition and an expectation agenda. In *Proceedings of Eurospeech*.

Bordes, A., and Weston, J. 2016. Learning end-to-end goal-oriented dialog. *CoRR* abs/1605.07683.

Bos, J.; Klein, E.; Lemon, O.; and Oka, T. 2003. DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture. In *SIGDIAL Workshop*, 115–124.

Ginzburg, J. 1996. Dynamics and the semantics of dialogue. In Seligman, J., and Westerståhl, D., eds., *Logic, Language and Computation*, volume 1. Stanford: CSLI.

Lison, P., and Kennington, C. 2016. OpenDial: A toolkit for developing spoken dialogue systems with probabilistic rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Demonstrations)*, 67–72.

Mittal, S.; Joshi, A.; and Finin, T. 2017. Thinking, Fast and Slow: Combining Vector Spaces and Knowledge Graphs. *arXiv:1708.03310 [cs]*. arXiv: 1708.03310.

Sonntag, D.; Nesselrath, R.; Sonnenberg, G.; and Herzog, G. 2009. Supporting a rapid dialogue system engineering process. In *Proceedings of the 1st IWSDS*.

Stoyanchev, S.; Lison, P.; and Bangalore, S. 2016. Rapid prototyping of form-driven dialogue systems using an open-source framework. In *Proceedings of the 17th Annual Meeting of the Special Interest Group on Discourse and Dialogue*.

Traum, D., and Larsson, S. 2003. The Information State Approach to Dialogue Management. In *Current and New Directions in Discourse and Dialogue*. 325–353.

Wessel, M.; Acharya, G.; Carpenter, J.; and Yin, M. 2017. An Ontology-Based Dialogue Management System for Virtual Personal Assistants. In *Proceedings of the 8th IWSDS*.

Williams, J. D.; Asadi, K.; and Zweig, G. 2017. Hybrid Code Networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. *arXiv preprint arXiv:1702.03274*.

Xu, W., and Rudnicky, A. I. 2000. Task-based dialog management using an agenda. In *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational Systems - Volume 3*, ANLP/NAACL-ConvSyst '00, 42–47.