An Application-Independent Approach to Building Task-Oriented Chatbots with Interactive Continual Learning

Sahisnu Mazumder, Bing Liu, Shuai Wang, Sepideh Esmaeilpour
Department of Computer Science, University of Illinois at Chicago, USA
sahisnumazumder@gmail.com,liub@uic.edu
shuaiwanghk@gmail.com,sesmae2@uic.edu

Abstract

Many real-life task-oriented chatbots are *natural language* (*command*) *interfaces* (NLIs) to their underlying applications. Such a NLI is often built using a semantic parser (SP) to parse the user command and convert it to a logical form, which is then translated to an executable action, or using an end-to-end deep learning model. They all need large volumes of application specific data for training. This paper proposes a *new and application independent* approach to building NLIs that needs no SP, translator, or training. It is based on NL to NL matching, where the representations of both actions and user commands are in natural language (NL). Given a user command, the system matches it with a correct action representation, and executes its associated action. The system can also continuously learn new NL expressions of actions from users through interactions to make the system more powerful. The learning happens during chatting after the model has been deployed. Our experimental results show the effectiveness of the proposed approach.

1 Introduction

Task-oriented chatbots like *virtual assistants* are essentially built as Natural Language (command) Interfaces (NLI) that allow users to issue natural language (NL) commands to be mapped to some actions for execution by the underlying application. Existing methods for building NLIs either train an end-to-end model [4, 42, 27, 13, 38] to map a NL command *directly* into an executable action or learns a semantic parser (SP) to parse the command into an intermediate logical form which is then translated into an executable action [47, 2, 1, 49, 20]. Despite their success in building NLIs, these approaches suffer from some key limitations: **(1)** they are trained with a large amount of application-specific training data which is very hard to come by, and no matter how large the data is used, it's hard to cover all possible language variations. If a SP approach is taken, for each application, a different translator is needed too. **(2)** For a different application, the models need to be retrained using an in-domain corpus. Due to these problems, it is hard to build an *application-independent system* that is not tied to application schema and can automatically adapt to different applications.

In this paper, we propose an entirely different approach- ***natural language to natural language*** (**NL2NL**) matching, to automatically build NLIs. The approach is *application-independent* and requires no pre-collected application-specific training data, and thus can be automatically adapted to different applications. This work focuses on NLIs for API (Application Programming Interface)-based applications such as robot navigation and command systems, virtual assistants like Siri and Alexa, GUI-based software applications (e.g., manipulating objects in MS Word, MS Paint, Windows), etc.

Problem statement: Let $\mathcal{A}$ be a set of actions in the form of APIs that can be performed in an application. If the user wants to perform an action, he/she issues a NL command C. Our goal is to

NeurIPS-2020 Workshop on Human in the Loop Dialogue Systems (HLDS-2020). 2020

match C to the right action $a_i \in \mathcal{A}$ to be executed. If the system fails to match, it learns from the user through active interactions. This continual learning makes the system more powerful over time.

Based on the NL2NL matching idea, we design a system, called **CML** (***Command Matching and Learning***) to automatically build NLIs for any API-driven applications. To build a new NLI (or incrementally add a new task/skill to an existing NLI), the application developer only needs to write a set S_i of *seed commands* (**SCs**) in NL to represent each API $a_i \in \mathcal{A}$. SCs in S_i are just like paraphrased NL commands from the end users to invoke a_i. The only difference is that the objects to be acted upon in each SC are replaced with variables, which are the arguments of API a_i. When the user issues a command C, the system simply matches C with a SC s_k^* of the correct action a^* and in doing so, it also instantiates the variables/arguments for the associated API a^* to be executed. To make the approach more powerful, we also introduce utility APIs and their SCs (see Sec. 3.2).

Let us use the *Microsoft Paint* tool and the API `drawCircle(`$X1$`,`$X2$`)` (drawing a circle having **color** $X1$ at **coordinate** $X2$) to illustrate. Let "*draw a* $X1$ *circle at* $X2$" be a SC for this API, where $X1$ and $X2$ are variables representing the arguments of the API. A user command "*draw a blue circle at (20, 40)*" can be matched or grounded to this SC, where the grounded API arguments are $X1$ = '*blue*' and $X2$ = *(20, 40)*.

Since the SCs written by the developer are not likely to cover all possible paraphrased expressions that a user may use to express the same command. This causes difficulty for matching. An *interactive learning mechanism* is proposed to enable CML to continually learn new (paraphrased) SCs from users so that when similar commands are issued in the future, the system can handle them easily.

CML has three key advantages: **(1)** Building a NLI for a new application requires no programming or training. **(2)** Since SCs are written in NL, the SC matching algorithm of CML is not tied to any application (API) schema and is thus application-independent. **(3)** CML can learn new SCs in the process of being used to make it more and more powerful. To our knowledge, no existing NLIs have all these advantages. Note that, CML is not a full-fledged task-oriented dialogue system as it does not have a *dialogue manager* responsible for *dialogue state tracking* and *dialogue policy* or perform multi-turn dialogues [7, 45]. However, it can form the core of an *application independent* task-oriented dialogue system as it provides the application independent *Natural Language Understanding* (NLU) functions of *intent identification* and *slot filling* [23, 50, 8]. Dialogue manager is relatively easy to build. We evaluate CML using three representative applications to demonstrate its effectiveness.

2 Related Work

NLIs has been used in diverse applications. For robot navigation, [2, 39, 40, 16, 12] proposed methods for grounding navigation instructions. For database querying NLIs [20], popular works include use of logic programming [47], combinatory categorial grammar [48]. Other works are [3, 46, 28, 18, 10, 21]. Besides, [32, 41, 17, 14] built NLIs for visual data analysis. For webpages and GUIs, [5] proposed a RL-based solution for mapping software troubleshooting guides to GUI actions. [37, 30] proposed end-to-end approaches for building NLI for web APIs. [34, 29] designed an NLI for submitting web forms and interacting with webpages and [22] proposed an NLI for Bash commands. Other works include selecting correct objects [11, 15, 33, 6] and discovering commands in multimodal interfaces [36]. All these approaches differ substantially from CML as they mostly learn models using pre-collected labeled data, which is hard to obtain. Also, unlike them, CML builds application-independent NLIs and only requires the developer to write SCs in NL for their APIs (which is easy to do). CML also continually learns new SCs from users to make it more powerful over time. For a general understanding of continual or lifelong learning, please refer to [9, 24].

CML is related to interactive language learning [44, 43], but unlike CML, they do not allow learner to acquire knowledge by asking questions to users in multi-turn NL dialogues. [19, 25] allow the learner to ask questions to users for factual knowledge learning. They are not about building NLIs.

3 Proposed Technique

The proposed system CML has three components: **(1)** an ***SC (seed command) specification*** (Sec 3.1), which enables the application developer to specify a set of SCs for each of their APIs ; **(2)** a ***command grounding module*** (Sec 3.2), that grounds a user command C to an action SC by matching C with the correct SC (whose associated action API is then executed), and **(3)** an ***interactive learner*** (Sec 3.3) that interacts with users to learn new SCs and paraphrases of API argument values.

Table 2: Action SC specifications for Blocks-World application and some example NL commands from user for each API. (*) denotes that the variable do not take part in command reduction (Utility Constraints), which is automatically detected and marked by CML (see Sec 3.2) (X denotes input).

Action API Function	AID	Action SCs (';' separated)	Variable: Argument Type	Example commands
AddBlock (X1)	**1**	add a block at X1; insert a block at X1	X1: 'location' (*)	add a block at (2, 3); put a block at (2, 3)
Remove (X1)	**2**	remove X1	X1: 'block_set'	delete blue block; take away blue block
Move (X1, X2)	**3**	move X1 to X2; shift X1 to X2	X1: 'block_set', X2: 'location' (*)	move blue block to the left of cube; shift green cube to (4, 5)
MoveByUnits (X1, X2, X3)	**4**	move X1 along X2 by X3 units	X1: 'block_set', X2: 'direction', X3: 'number'	move blue block left by 2 units; shift green cube down by 3 units
UpdateColor (X1, X2)	**5**	change color of X1 to X2; color X1 with X2	X1:'block_set', X2: 'color' (*)	color A red; change color of B to blue
UpdateShape (X1, X2)	**6**	change shape of X1 to X2	X1:'block_set', X2: 'shape' (*)	set the shape of A to cube; make B square
Rename (X1, X2)	**7**	rename block X1 to X2	X1: 'block_set', X2: 'name' (*)	Name the block at (4, 5) as C; rename A to D

Table 3: Utility SC specifications for Blocks-World & example referential expressions (O: output, X: input)

Utility API Function	AID	Utility SCs (';' separated)	Variable: Argument Type	Example referential expressions
GetBlocksbyColor(X1)	**8**	X1 block; the X1 one	X1: 'color', O1: 'block_set'	blue block; the green one
GetBlocksbyShape(X1)	**9**	X1 block; the X1 one	X1: 'shape', O1: 'block_set'	square block; the round one
GetBlocksbyName(X1)	**10**	X1; block marked X1	X1: 'name', O1: 'block_set'	A; block marked B
GetBlocksAtLocation(X1)	**11**	block at X1	X1: 'location', O1: 'block_set'	block at row 2 and column 3
GetLocation(X1, X2)	**12**	X1 of X2	X1: 'direction', X2: 'block_set', O1: 'location'	left of blue block; below block B

In our problem setting, we assume an application comes with two sets of APIs: **action APIs** $\mathcal{A}$ and **utility APIs** $\mathcal{U}$. $\mathcal{A}$ defines a set of actions that can be performed in the application. Each action $a_i \in \mathcal{A}$ causes a change in the *state* of the (instantiated) objects in the application, specified by objects' properties and their values. E.g., in *Microsoft Paint*, a circle drawn on the editor is an example of an *instantiated* object and it can have properties like *color*, *name*, *shape*, etc, with their values like the color *red*. Examples of actions are: *draw a circle*, *change the shape of the circle* etc.

Utility APIs in $\mathcal{U}$ are used to resolve *referential expressions* in user command. E.g., in the user command: "*move the blue block to the left of the cube*", "*blue block*", "*cube*", "*left of the cube*" are referential expressions, where "*blue block*" refers to some block having color "*blue*"; "*cube*" refers to some block having shape "*cube*" and "*left of the cube*" refers to some *location* in the application space, that is in the left of the cube shaped block and where the blue block is to be moved to. Utility APIs help in resolving referential expressions by identifying the referred objects (by object ids) having a given property value or a property value referred through an object in current state of the application. The resulting objects or property value are used by action APIs and/or other utility APIs to unambiguously ground the user command (discussed in Sec 3.2).

Table 1: Properties and their domains for Blocks-World

Property	Domain
color (object)	'red', 'green', 'orange', 'blue', 'yellow'
shape (object)	'triangular', 'circular', 'cube', 'square', 'rectangular'
location (object)	(x, y) coordinate in 2D space
name (object)	denoted by any letter from A-Z
direction (action)	'right', 'left', 'above', 'below'

3.1 SC Specification

SC specification consists of three parts: **(1)** *Properties and their domains*, **(2)** *Action SC specification*, and **(3)** *Utility SC specification*. We use a Blocks-World application to illustrate the specification. The application is about arranging different blocks (objects) of different *shapes*, *colors* and *names* on a 2D grid or inserting them to form a goal arrangement using some action APIs.

Properties and Domains: Table 1 shows different properties of an object or action (specified in the bracket next to the property name) and their domains. E.g., "*color*" is a property of a block with domain values '*red*', '*green*' etc. and '*direction*' specifies execution of an action in a given direction.

Action and Utility SC Specifications: Table 2 and 3 show seven action APIs and five utility APIs respectively used in our Block-World application, along with some example user commands and referential expressions that can fire them. The argument type '*block_set*' denotes a set of block ids. Table 2 and 3 also show an unique API ID (AID) for each (action or utility) API which we use to refer them in the following sections. The arguments marked (*) in Table 2 denote utility constraints automatically marked by CML to restrict the use of utility SCs (discussed in Sec. 3.2).

3.2 Command Grounding Module

Given a user command C and the SC specification, the ***command grounding module*** (**CGM**) returns a grounded SC set $\hat{A}$, consisting of one *action* SC and a set of *utility* SCs, telling the system what action to perform. If the grounding is not successful, $\emptyset$ is returned. **Algorithm 1** shows the grounding process. We still use the Blocks-World application (Figure 1) to explain. CGM has two main modules:

(i) Rephraser and Tagger ($\mathcal{R}$) [Line 1, Algorithm 1]: Given the user command C, $\mathcal{R}$ repharses C and tags each word or phrase in the *rephrased* C with either 'O' (i.e., not an argument type) or one of the possible argument types of the action SCs. This work uses a dictionary lookup and regular expression-based rephraser and tagger for $\mathcal{R}$. In this process, $\mathcal{R}$ first reads the SC specification and forms a tagset by enumerating all *argument types* in the action SCs and also, builds a *repharser dictionary* with (*argument type*, *argument domain value*) pair (e.g. *shape* = *circular*) as key and a list of paraphrases (e.g. {'*round*', '*disk like*', '*ring shaped*'}) of the domain value (i.e, "*circular*") as the corresponding dictionary value of the key. We obtain the synonyms of the domain values from WordNet [26] and ConceptNet [35] to populate the dictionary and use it to rephrase C by replacing words and/or phrases with property domain values (i.e., the keys). While rephrasing, the phrases in C can be directly tagged with argument types using the dictionary. For example, the user command C in Figure 1 is rephrased as "*relocate the **circular** block to the left of D*", where "*round*" is replaced with synonym domain value "*circular*" and then, tagged as "*relocate the **shape/X1** block to the **direction/X2** of **name/X3***", i.e., C'_0 in Figure 1 [where, *X1*="*circular*", *X2*="*left*", *X3*="*D*"].

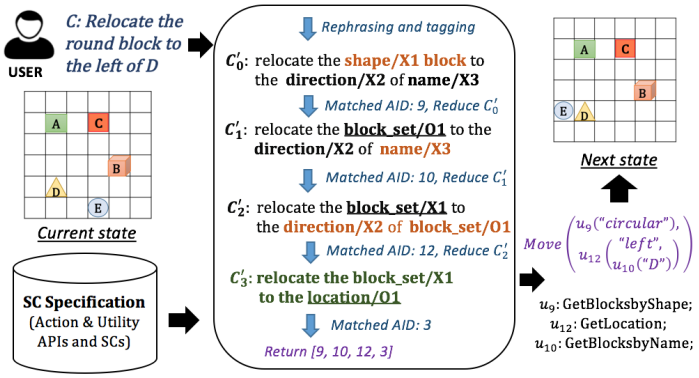

Figure 1: Working of CGM on a user command for Blocks-World. AID denotes the API IDs (see Tables 2 and 3).

(ii) *SC Matcher* [or simply *Matcher*] ($\mathcal{M}$): Given the rephrased and tagged command C' and the set $\mathcal{T}$ of (action or utility) SCs, Matcher $\mathcal{M}$ computes a match score $f(t, C')$ for each $t \in \mathcal{T}$ and returns the top ranked SC $\hat{t} = \arg\max_{t \in \mathcal{T}} f(t, C')$. Any paraphrasing model can be used as $\mathcal{M}$. This work uses information retrieval (IR) based unsupervised matching models for $\mathcal{M}$ (see comparison in the Experiment section). Below, We discuss the steps (Lines 2-24) in Algorithm 1.

Initialization (Lines 2-3): CGM first enumerates and extracts the list E_{sub} of ***sub-expressions*** (SE) from C' (Line 2) to assist iterative grounding of C' using the utility SCs. A sub-expression of length-m is a substring of C' involving m consecutive variables (with types) and intermediate words linking them. For example, the list of all SEs in C'_0 of Figure 1: E_{sub} = ["*shape/X1*", "*direction/X2*", "*name/X3*", "*shape/X1 block to the direction/X2*", "*direction/X2 of name/X3*"], where the first three in E_{sub} are length-1 and last two are length-2 SEs. Length-3 SE is the full command C'_0 and is discarded from E_{sub} as it is matched with action SCs rather than the utility SCs.

Iterative Grounding Steps [Lines 4-24]: With the resulting E_{sub}, C' is grounded as follows:

- First, a candidate set A_{set} of action SCs is retrieved from $\mathcal{T}$ (SC store) such that for any $a \in A_{set}$, C' and a has an ***identical set of variables and types***. If $A_{set} \neq \emptyset$, $\mathcal{M}$ returns the top ranked action SC a_k [Lines 21-22] for C' and the loop terminates. In Figure 1, none of the action SCs are matched directly for C'_0 and so, $A_{set} = \emptyset$.
- If $A_{set} = \emptyset$, Matcher $\mathcal{M}$ works as follows: If $E_{sub} = \emptyset$ or all SEs in E_{sub} have been checked, it returns $\emptyset$ indicating C is ungroundable [Lines 7-8], i.e., no action SCs in $\mathcal{T}$ can be matched. Otherwise, it selects the sub-expressions from E_{sub} one by one, reduces C' further by resolving bindings for the variables in the sub-expression and then, attempts to match the reduced C' with action SCs in each iteration, until a match is found (Lines 5 and 21-23) or the condition of ungroundablity (in Lines 6-7) is satisfied.

The resolving of a sub-expression (say, e_j) and successive reduction of C' [Lines 10-14] works as follows: Matcher first selects a candidate set U_{set} of utility SCs such that for any $u \in U_{set}$, e_j and

u has ***identical set of variables and their types*** [Line 10]. Note, while matching the variables with utility SCs, we **rename the variables in** e_j to avoid error in matching due to different variable names. E.g., variable X3 in sub-expression "*name/X3*" [see C'_1 in Figure 1] is renamed as X1 [i.e."*name/X1*"] (not shown), so that it can match with AID 10, while reducing C'_1 to C'_2.

• If $U_{set} = \emptyset$, Matcher $\mathcal{M}$ cannot perform reduction of C' for e_j and only j is incremented [Line 18]. Otherwise, $\mathcal{M}$ returns the top ranked utility SC u_r for e_j [Line 12]. Next, C' is reduced by replacing e_j with "*type*(O1)/O1" [the output variable and its argument type corresponding to u_r] (Line 14). E.g., given e_j ="*name/X3*" [see C'_1 in Figure 1], utility AID 10 gets matched and C'_1 gets reduced to C'_2, where "O1" is the output variable and $type(O1)$= "*block_set*" [i.e., set of block ids]. As a part of reduction, the variables of C' obtained after replacement are also renamed [i.e. from *block_set/O1* in C'_2 in Figure 1 to *block_set/X3* (not shown)] and aliases are recorded, so that in the next iteration, it can be matched with the action SCs [in Lines 5 and 21]. After reduction, we again extract new E_{sub} using the new (reduced) C' and set j to 0 (Lines 15-16) similar to Line 2. CGM also stores the values of the output variables in a buffer obtained by executing the matched utility SCs for subsequent grounding. In Figure 1, C'_3 matches with action API of AID 3 and the iterative grounding completes here. The final grounded AID list in Figure 1 is $[9, 10, 12, 3]$.

Algorithm 1 Iterative Command Grounding

Input: C: NL command issued by the user; $\mathcal{T}$: SC store containing all (action and utility) SCs; $\mathcal{R}$: Rephraser and tagger module; $\mathcal{M}$: SC Matcher;
Output: $\hat{A}$: predicted AID set for grounding C;

1: $C' \leftarrow$ **Rephrasing_and_Tagging**(C, $\mathcal{R}$)
2: $E_{sub} \leftarrow$ **Extract_SEs**(C', $\mathcal{T}$, $\mathcal{M}$)
3: $\hat{A} \leftarrow \emptyset$; $j \leftarrow 0$
4: **while** $TRUE$ **do**
5: $A_{set} \leftarrow$ **GetCandActionSCs**($\mathcal{T}$, C')
6: **if** $A_{set} = \emptyset$ **then**
7: **if** $E_{sub} = \emptyset$ or $j > |E_{sub}| - 1$ **then**
8: **return** $\emptyset$
9: **else**
10: $U_{set} \leftarrow$ **GetCandUtilitySCs**($\mathcal{T}$, e_j) { here, $e_j = E_{sub}[j]$ }
11: **if** $U_{set} \neq \emptyset$ **then**
12: $u_r \leftarrow$ **GetTopRankedSC**(U_{set}, e_j, $\mathcal{M}$)
13: $\hat{A} \leftarrow \hat{A} \cup \{u_r\}$
14: $C' \leftarrow$ **ReduceCommand**(C', e_j, u_r)
15: $E_{sub} \leftarrow$ **Extract_SEs**(C', $\mathcal{T}$, $\mathcal{M}$)
16: $j \leftarrow 0$
17: **end if**
18: $j \leftarrow j + 1$
19: **end if**
20: **else**
21: $a_k \leftarrow$ **GetTopRankedSC**(A_{set}, C', $\mathcal{M}$)
22: **return** $\hat{A} \cup \{a_k\}$
23: **end if**
24: **end while**

During extraction of sub-expressions from C' (in Lines 2 and 15), we also filter out sub-expressions that are marked with utility constraint markers [(*) in Table 2] so that they are not reduced by utility SCs as discussed below.

Utility Constraint Marker. In some cases, some variables (arguments) in the user command should not be reduced by the utility SCs as it can result in incorrect subsequent grounding. Consider the SC with AID 1 in Table 2 for adding a block at location X1. Here, the argument *X1:'location'* [marked (*)] should not be resolved using utility AID 11 (getting the block(s) at a given location). Otherwise, it will wrongly reduce a user command like "add a block at (2, 3)" to "add a block at block_set/X1". CML marks these action SC arguments to indicate no reduction should be applied.

Let $u_{seq} = \langle type(X1_u), ..., type(XN_u) \rangle$ be the sequence of variable types in a utility SC u from left to right, where Xi_u is the ith variable in u. Similarly, let a_{seq} be the sequence of variable types in an action SC a. Let M_{seq} be the longest common sub-sequence of u_{seq} and a_{seq}. Then if $|M_{seq}| = |u_{seq}|$ ($|\cdot|$ denotes length), all variables corresponding to the argument types in M_{seq} should be marked with (*) in a to indicate utility constraints for action SC a. Note, if $|M_{seq}| = |u_{seq}|$, a sub-expression involving M_{seq} will fire u, causing reduction of the user command using u and so, the reduced command cannot be matched with a in subsequent steps, leading to error in grounding.

To filter out SEs, CGM first uses Matcher $\mathcal{M}$ to identify the top-ranked action SC semantically close to C' [e.g., AID 1 for the user command "add a block at **(2, 3)**"] and then, delete all SEs involving such arguments (e.g., "*location/X1*") from E_{sub} extracted from C'.

3.3 Interactive Learner

As a given SC can have many paraphrased versions and the developer can only provide a limited number of them, CML may not find significant match for an unseen paraphrased user command. Thus,

we enable CML to learn new SCs from users through interactions. Additionally, CML learns new paraphrases of argument (domain) values from users to improve $\mathcal{R}$ over time. **Algorithm 2** shows the overall interactive learning process. Note, all questions asked by CML as part of the interaction (in Algo-2) are formulated based on a fixed NL question templates set $\mathcal{Q}$ (see Supplementary).

Once the user command C is grounded into some action SC (i.e., Line 22, Algo-1) or the grounding fails (i.e., Line 8, Algo-1), CML shows the user the predicted action SC with the detected argument values or a grounding failure message for verification[1] (Line 1, Algo-2). Here, a SC verification question is asked to the user (e.g. "*Am I correct?[yes/No]*") and a yes/no answer is expected. If the user says "yes", the system executes the command by invoking the corresponding action API. Otherwise, CML proceeds to learn new knowledge from the user as follows (Lines 2-12, Algo-2).

If the user says "No", CML shows a ranked list C_{rnk} of action SCs and asks the user to choose the correct option by scrolling the list [Lines 2-4, Algo 2]. Here, C_{rnk} is generated by scoring the action SCs (using $\mathcal{M}$) and selecting the top-ranked SC for each action API. C_{rnk} also include a "*none of these*" option for the user to acknowledge that C is ungroundable.

Algorithm 2 Interactive Knowledge Learning

Input: C': Reduced user command by Algorithm 1; $\mathcal{T}$: action and utility SC Store; $\mathcal{Q}$: Question Template Store; $\mathcal{M}$: SC Matcher;

1: $r_1 \leftarrow$ **Verify_Pred_SC**($\mathcal{Q}$, C') {r_i is user's response}
2: **if** $r_1 =$ "no" **then**
3: $r_2 \leftarrow$ **ShowSC_List**(C_{rnk}) {C_{rnk} is the action SC rank list returned by $\mathcal{M}$}
4: **end if**
5: **for all** variable x_i in r_2 **do**
6: $r_{expr} \leftarrow$ **Ask_Ref_Expr**(x_i, C)
7: $r_{prop} \leftarrow$ **Ask_Ref_Prop**(r_{expr})
8: $r_{val} \leftarrow$ **Choose_Prop_Val**(r_{prop}, r_{expr})
9: $r_{para} \leftarrow$ **Ask_Para_Expr**(r_{val}, r_{expr})
10: Update $\mathcal{R}$ with all (r_{val}, r_{para}) pairs
11: **end for**
12: Rephrase C to get a new SC and update $\mathcal{T}$

After user chooses the correct SC (r_2 in Algo-2), CML proceeds to acquire the referential expressions and argument value paraphrases in C for each API argument of the chosen SC (Lines 5-10, Algo-2), in subsequent dialogue turns. The muti-turn dialogues for each argument x_i proceeds as follows: First, CML asks user to provide referential expression r_{expr} for x_i in C [Line 6]. Next, it asks to list all properties referred in r_{expr} [Line 7]. Then, for each property r_{prop} listed by the user for r_{expr}, CML asks the user to choose the property value(s) [from domain of r_{prop}] which user has mentioned in r_{expr} [Line 8]. Finally, for each property value r_{val} of a given property r_{prop}, CML asks the user to denote the paraphrased NL expression r_{para} in r_{expr} that represents r_{val} [Line 9]. The (r_{val},r_{para}) pairs are used to update dictionary of $\mathcal{R}$ [Line 10]. Note, in this process, if some r_{expr} or r_{para} or r_{val} can be directly detected by CML via dictionary lookup using $\mathcal{R}$, corresponding dialogue turn is skipped.

Once all x_i(s) are processed, CML replaces r_{expr}(s) in C with corresponding x_i(s) to transform C into a new SC for r_2 and update $\mathcal{T}$ [Line 11]. See Table 7 Supplementary for example dialogue.

4 Experiments

We evaluate CML on three representative applications: (1) *Blocks-World* (BW), (2) *Webpage Design* (WPD) and (3) *Flight Booking* (FB). To conduct evaluation in an interactive setting, we create a **simulated user** (a program) for each application that issues NL commands to CML from a (labeled) ***test dataset*** ($\mathcal{D}$), provides feedback to CML's predicted API set and answers CML's questions for learning new SCs and paraphrases of argument values (Algo-2) using a ***knowledge base*** ($\mathcal{K}_u$). For applications (1) and (2), we build $\mathcal{D}$ and $\mathcal{K}_u$ of the simulated user by collecting an annotated command dataset written by human users. For (3), we adopt a subset of ATIS NLU bechmark dataset[2] to populate $\mathcal{D}$ and $\mathcal{K}_u$ (see Supplementary Material).

For collecting the command dataset, we showed the supported API functions of the application to five human users (graduate students, who are unaware of the working of CML) and asked them to write commands to play with the application. For each command, we also asked these five human users to write down the gold set of APIs. Also, for each gold action API (for a given NL command), we asked them to provide the gold argument values along with phrases in the NL command denoting these values along with the gold action SC. Given this compiled annotated data, we use the NL commands

[1]The verification step is mainly required for critical applications with safety concerns. Otherwise, for an incorrectly executed task, user can say "*undo*" to revert changes and initiate dialogue for interactive learning.

[2]github.com/howl-anderson/ATIS_dataset

written by the human users to create the *test dataset* ($\mathcal{D}$) and store the gold SCs and gold argument values and their paraphrases for each argument corresponding to the gold action SC in simulated user's knowledge base ($\mathcal{K}_u$). Note, as CML can only ask a fixed set of questions, we can build $\mathcal{K}_u$ with full coverage of CML query set with the annotated gold dataset.

The list of test commands are randomly shuffled and fed to CML by the simulated user one by one for grounding. Once CML makes the prediction, user provides feedback about the correctness of the grounded action API, which is used to compute the evaluation metrics. In particular, we report the *action API accuracy* (**A-acc**) score to measure the correctness of action API prediction and avg. *argument F1* (**Arg-F1**) score (computed using avg. argument precision and avg. argument recall over test commands) to measure the performance on argument types prediction. For a given test command, argument precision denotes the fraction of the predicted list of argument types that are correct and argument recall is the fraction of the (gold) list of argument types predicted for the command. While collecting test commands, we also ask human users to write some commands that are *ungroundable*, for which the gold API set is $\emptyset$ (not groundable to any of the action SCs (or APIs)). Thus, if CML returns $\emptyset$ for such commands, it is considered correct. Table 4 shows the test command dataset ($\mathcal{D}$) statistics. The **SC specifications** for the Blocks-World (BW) applications were given in Tables 2 and 3 and those for Webpage Design (WPD) and Flight Booking (FB) are given in the Supplementary.

Table 4: Dataset statistics. n-UC denotes the number of user commands that needs n utility SCs for grounding. "UG" (ungroundable) denotes the number of commands that are unungroundable. For flight booking (FB), the NL commands can directly be grounded to action APIs without the need to resolve referential expressions using utility APIs (Lines 7-19, Algo-1). Thus, all commands fall into 0-UC category.

Application	0-UC	1-UC	2-UC	> 2-UC	UG	Total
BW	15	174	22	88	39	338
WPD	14	170	14	23	14	235
FB	1183	-	-	-	-	1183

Compared Models. As there is no existing work using the NL2NL approach, we compare various versions of the CML model (2-8 in the list below). We don't compare with existing parsing and/or end-to-end methods as they need application-specific training data and are not application independent. However, as we adopt the ATIS NLU dataset for our evaluation, we also compare a modified version of a recent NLU model [described in (1)] in an identical evaluation setting as used for the CML variants (2-8) below.

Table 5: Performance comparison of CML variants and BERT-JISF. Here, CML-vsm(-U) and CML-vsm results are the same for FB as utility APIs are absent in FB specifications].

Models	BW		WPD		FB	
	A-acc	Arg-F1	A-acc	Arg-F1	A-acc	Arg-F1
BERT-JISF	49.70	59.23	57.87	58.17	34.06	57.61
CML-jac	**68.93**	79.35	74.04	83.27	86.22	**97.35**
CML-vsm	**68.93**	79.35	**74.46**	**85.81**	**88.08**	**97.35**
CML-embed	68.63	**79.94**	68.93	82.83	83.43	**97.35**
CML-vsm (-R)	64.79	77.42	68.08	79.43	76.58	88.35
CML-vsm (-U)	14.49	14.49	11.48	11.70	-	-
CML-jac + SCL	69.82	**81.34**	76.17	84.43	92.05	**97.35**
CML-vsm + SCL	**70.11**	80.09	**77.02**	**85.49**	**93.32**	**97.35**
CML-jac + SCL + APL	72.78	**81.43**	80.0	85.31	93.15	**98.41**
CML-vsm + SCL + APL	**73.07**	80.16	**80.85**	**86.37**	**94.67**	**98.41**

(1) BERT-JISF: This is a state-of-the-art joint intent detection and slot filling model that finite-tunes a pre-trained BERT model[3] to solve NLU [8]. In our setting, we treat the action APIs as intents and argument types as slots. As we don't use any explicit application training data for building NLI, we train BERT-JISF using a labelled data generated by instantiating the initial SCs (provided by the devloper) by randomly sampling argument values from corresponding domains and replacing the variables in SCs with those values (see Supplementary for more details). The model is evaluated against the same test set (in Table 4) as used for the CML variants. **(2) CML-jac:** Jaccard similarity is used as the scoring function of *Matcher* $\mathcal{M}$. **(3) CML-vsm:** Tf-idf based vector space model is used for $\mathcal{M}$ where all SCs associated with each API are regarded as one document and the user command as the query. **(4) CML-emb:** Word embedding based matching model is used for $\mathcal{M}$. Given an SC t and a user command (after tagging) C', we retrieve the pre-trained word embedding vector for each word in C' (t) and average them to get the vector representation of C' (t) as $v_{c'}$ (v_t). Next, we use cosine similarity as the scoring function to measure the similarity between $v_{c'}$ and v_t. We use 50D Glove [31] embeddings for evaluation. **(5) CML-vsm(-R):** Variant of CML-vsm, where the rephrasing of words in the input command [Line 1, algo-1] is disabled. **(6) CML-vsm(-U):** Variant of CML-vsm, where the use of utility SCs while grounding [i.e., Lines 7-19, algo-1] is disabled. For (5) and (6), although we only report performance of CML-vsm variants here, we also compared these

[3]https://tfhub.dev/google/bert_uncased_L-12_H-768_A-12/1

variants for CML-emb and CML-jac and found their results are poorer. For these 5 CML variants, we disabled the SC Learner for learning new SCs. **(7) CML-Y + SCL:** variants of CML-Y (Y is *jac*, *vsm*, or *emb*) that can learn *new SCs* through user interactions, but learning of new *new paraphrases of argument values* is disabled (Lines 9, Algo-2). **(8) CML-Y + SCL + APL:** variants of CML-Y (Y is *jac*, *vsm*, or *emb*), where we allow CML-Y to learn **both** *new SCs* and *paraphrases of argument values* through user interactions. Since the emb version is relatively poor (discussed next), we chose to compare the *jac* and *vsm* variants for (7), (8).

4.1 Results and Analysis

Table 5 shows the performance comparison of CML variants and BERT-JISF on three applications. CML-jac and CML-vsm perform better overall. The drop in performance for CML-emb in WPD and FB is primarily due to the out of vocabulary words (no pre-trained embeddings) in user commands. The performance of CML-vsm(-R) drops significantly, which shows that rephrasing helps greatly in command reduction and grounding. CML-vsm(-U) performs the worst among all variants which shows the importance of user command reduction using utility SCs. The poor performance of BERT-JISF is due to the limitation in linguistic richness in instantiated training data (from seed SCs) which is not sufficient to generalize well. This also shows, the model need large data sets with sufficient linguistic variations for better generalization as used in traditional NLU works [8] whereas CML can achieve impressive performance without need for any application-specific training.

As CML-jac and CML-vsm perform the best overall for all three applications, we also compare the performance of the *interactive knowledge learning* variants (i.e., CML-Y+SCL and CML-Y+SCL+APL) of these two CML versions in Table 5. We can see that SCL variants (i.e., CML-Y+SCL) clearly outperforms non-SCL variants (i.e., CML-jac and CML-vsm) and SCL+APL variants improve performance over corresponding SCL only variants for three applications. It is very important to note that these improvements are gained from the existing datasets, which do not have many similar commands to the newly learned SCs. In practice, if similar commands are repeated by many users, the improvement will grow substantially. The SCL+APL variants additionally learn new paraphrases of argument values which enhances the argument tagging capability of CML, resulting in substantial performance improvements. The performance improvements (A-acc) of SCL variants for WPD and FB are more than that for BW, which can be explained as follows. For BW, the arguments and their types in API and SC specifications (see Tables 2 and 3) are quite distinguishable from each other. Thus, correctly identifying arguments in user commands play a major role in the success of command grounding. Learning of new action SCs does not make significant impact here. But, for WPD (see Supplementary), action SCs with AIDs 8, 9 and 10 have exactly the same arguments and types, but they differ significantly in action intents. Thus, learning new SCs helps greatly, which also explains the gain for FB. To investigate the effect of SC learning further, we evaluate CML-jac and CML-jac + SCL (CML-vsm and CML-vsm + SCL) on user commands that are only groundable to any of the action APIs with AIDs 8, 9 and 10 for WPD. Here, we observe almost 4% and 13% improvement in A-acc for SCL variants of CML-jac and CML-vsm respectively, which justifies the explanation.

Table 6: Accuracy gains of SC Learner variants on user commands groundable to actions with AIDs 8, 9 and 10 in Webpage Design (see Supplementary)

Models	A-acc
CML-jac	75.0
CML-vsm	70.83
CML-jac + SCL	**79.16**
CML-vsm + SCL	**83.33**

Table 7: Accuracy (A-acc) for various command categories. Here, UG denotes "Ungroundable".

App	0-UC	1-UC	2-UC	>2-UC	UG
BW	66.66	82.18	36.36	60.22	48.71
WPD	92.85	78.23	57.14	47.82	71.42

In Table 7, we compare CML-vsm over various command types (listed in Table 4). We see that for 0-UC and 1-UC, CML-vsm performs significantly better than that for 2-UC, >2-UC and UG (note, for UG user commands, the gold API set is considered as $\emptyset$) as these commands are harder to ground due to the requirement of multiple (recursive) reduction steps using the utility SCs.

5 Conclusion

This paper proposed an natural language to natural language (NL2NL) approach to building natural language interfaces and a system CML, which are very different from traditional approaches. CML is application independent except that the SCs need to be specified by the application developer. CML can also learn new SCs and paraphrases of argument values interactively and continually from the end users over time to make it more powerful. Our evaluation showed that CML is highly promising. Our future work will study action compositions for more complex applications.

Acknowledgments

This work was supported in part by a research gift from Northrop Grumman, a research contract from DARPA (HR001120C0023), and two research grants from National Science Foundation (IIS-1910424 and IIS-1838770).

References

[1] Jacob Andreas and Dan Klein. Alignment-based compositional semantics for instruction following. In *EMNLP*, 2015.

[2] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. MIT Press, 2013.

[3] Christopher Baik, HV Jagadish, and Yunyao Li. Bridging the semantic gap with sql query logs in natural language interfaces to databases. In *ICDE*, 2019.

[4] Satchuthananthavale RK Branavan, Harr Chen, Luke S Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *ACL-IJCNLP*, 2009.

[5] SRK Branavan, Luke S Zettlemoyer, and Regina Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *ACL*, 2010.

[6] Asli Celikyilmaz, Zhaleh Feizollahi, Dilek Hakkani-Tur, and Ruhi Sarikaya. Resolving referring expressions in conversational dialogs for natural user interfaces. In *EMNLP*, 2014.

[7] Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. A survey on dialogue systems: Recent advances and new frontiers. *Acm Sigkdd Explorations Newsletter*, 2017.

[8] Qian Chen, Zhu Zhuo, and Wen Wang. Bert for joint intent classification and slot filling. *arXiv preprint arXiv:1902.10909*, 2019.

[9] Zhiyuan Chen and Bing Liu. Lifelong machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2018.

[10] Sébastien Ferré. Sparklis: an expressive query builder for sparql endpoints with guidance in natural language. *Semantic Web*, 2017.

[11] Michael C Frank and Noah D Goodman. Predicting pragmatic reasoning in language games. *Science*, 2012.

[12] Daniel Fried, Jacob Andreas, and Dan Klein. Unified pragmatic models for generating and following instructions. In *NAACL-HLT*, 2018.

[13] Daniel Fried, Ronghang Hu, Volkan Cirik, Anna Rohrbach, Jacob Andreas, Louis-Philippe Morency, Taylor Berg-Kirkpatrick, Kate Saenko, Dan Klein, and Trevor Darrell. Speaker-follower models for vision-and-language navigation. In *Advances in Neural Information Processing Systems*, 2018.

[14] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G Karahalios. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *UIST*, 2015.

[15] Dave Golland, Percy Liang, and Dan Klein. A game-theoretic approach to generating spatial descriptions. In *EMNLP*, 2010.

[16] Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *ACL*, 2017.

[17] Carolin Lawrence and Stefan Riezler. Nlmaps: A natural language interface to query open-streetmap. In *COLING*, 2016.

[18] Jingjing Li, Wenlu Wang, Wei-Shinn Ku, Yingtao Tian, and Haixun Wang. Spatialnli: A spatial domain natural language interface to databases using spatial comprehension. In *ACM SIGSPATIAL*, 2019.

[19] Jiwei Li, Alexander H Miller, Sumit Chopra, Marc'Aurelio Ranzato, and Jason Weston. Learning through dialogue interactions by asking questions. *arXiv preprint arXiv:1612.04936*, 2016.

[20] Yunyao Li and Davood Rafiei. Natural language data management and interfaces. Morgan & Claypool Publishers, 2018.

[21] Percy Liang. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, 2016.

[22] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *LREC*, 2018.

[23] Bing Liu and Ian Lane. Attention-based recurrent neural network models for joint intent detection and slot filling. *Interspeech*, 2016.

[24] Bing Liu and Sahisnu Mazumder. Lifelong learning dialogue systems: chatbots that self-learn on the job. *To appear in AAAI (senior member track)*, 2021.

[25] Sahisnu Mazumder, Bing Liu, Shuai Wang, and Nianzu Ma. Lifelong and interactive learning of factual knowledge in dialogues. In *SIGDIAL*, 2019.

[26] George A Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J Miller. Introduction to wordnet: An on-line lexical database. *International journal of lexicography*, 1990.

[27] Dipendra Misra, John Langford, and Yoav Artzi. Mapping instructions and visual observations to actions with reinforcement learning. In *EMNLP*, 2017.

[28] Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. In *ICLR*, 2017.

[29] Panupong Pasupat, Tian-Shun Jiang, Evan Liu, Kelvin Guu, and Percy Liang. Mapping natural language commands to web elements. In *EMNLP*, 2018.

[30] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *ACL*, 2015.

[31] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[32] Vidya Setlur, Sarah E Battersby, Melanie Tory, Rich Gossweiler, and Angel X Chang. Eviza: A natural language interface for visual analysis. In *UIST*. ACM, 2016.

[33] Nathaniel J Smith, Noah Goodman, and Michael Frank. Learning and using language via recursive pragmatic reasoning about other agents. In *NIPS*, 2013.

[34] K. Soh. Tagui: Rpa / cli tool for automating user interactions. In *https://github.com/ kelaberetiv/TagUI*, 2017.

[35] Robert Speer, Joshua Chin, and Catherine Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge. In *AAAI*, 2017.

[36] Arjun Srinivasan, Mira Dontcheva, Eytan Adar, and Seth Walker. Discovering natural language commands in multimodal interfaces. In *IUI*, 2019.

[37] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. Building natural language interfaces to web apis. In *CIKM*, 2017.

[38] Stefanie Tellex, Nakul Gopalan, Hadas Kress-Gazit, and Cynthia Matuszek. Robots that use language. *Annual Review of Control, Robotics, and Autonomous Systems*, 2020.

[39] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R Walter, Ashis Gopal Banerjee, Seth Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*, 2011.

[40] Jesse Thomason, Shiqi Zhang, Raymond J Mooney, and Peter Stone. Learning to interpret natural language commands through human-robot dialog. In *IJCAI*, 2015.

[41] Prasetya Utama, Nathaniel Weir, Fuat Basik, Carsten Binnig, Ugur Cetintemel, Benjamin Hättasch, Amir Ilkhechi, Shekar Ramaswamy, and Arif Usta. An end-to-end neural natural language interface for databases. *arXiv preprint arXiv:1804.00401*, 2018.

[42] Adam Vogel and Dan Jurafsky. Learning to follow navigational directions. In *ACL*, 2010.

[43] Sida I Wang, Samuel Ginn, Percy Liang, and Christoper D Manning. Naturalizing a programming language via interactive learning. In *ACL*, 2017.

[44] Sida I Wang, Percy Liang, and Christopher D Manning. Learning language games through interaction. In *ACL*, 2016.

[45] Tsung-Hsien Wen, David Vandyke, Nikola Mrkšić, Milica Gasic, Lina M Rojas Barahona, Pei-Hao Su, Stefan Ultes, and Steve Young. A network-based end-to-end trainable task-oriented dialogue system. In *EACL*, 2017.

[46] Hongvu Xiong and Ruixiao Sun. Transferable natural language interface to structured queries aided by adversarial generation. In *ICSC*. IEEE, 2019.

[47] John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programing. In *AAAI*, 1996.

[48] Luke Zettlemoyer and Michael Collins. Online learning of relaxed ccg grammars for parsing to logical form. In *EMNLP*, 2007.

[49] Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint*, 2012.

[50] Xiaodong Zhang and Houfeng Wang. A joint model of intent determination and slot filling for spoken language understanding. In *IJCAI*, 2016.