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
sahasnumazumder@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 (NLIs) 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 [1, 4, 27, 13, 58] 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, 41, 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 \( A \) 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
match \( C \) to the right action \( a_i \in A \) 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 \( a_i \in A \). 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 \( a^*_i \) and in doing so, it also instantiates the variables/arguments for the associated API \( a^*_i \) 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 \( \text{drawCircle}(X_1, X_2) \) (drawing a circle having color \( X_1 \) at coordinate \( X_2 \)) to illustrate. Let “draw a \( X_1 \) circle at \( X_2 \)” be a SC for this API, where \( X_1 \) and \( X_2 \) 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 \( X_1 = \text{'blue'} \) and \( X_2 = (20, 40) \).

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][25]. 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 [22][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][15] 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][28] 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).

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

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

<table>
<thead>
<tr>
<th>Utility API Function</th>
<th>AID</th>
<th>Utility SCs (’,’ separated)</th>
<th>Variable: Argument Type</th>
<th>Example referential expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBlockbyColor(X1)</td>
<td>8</td>
<td>X1 block; the X1 one</td>
<td>X1: ‘color’, O1: ‘block_set’</td>
<td>blue block, the green one</td>
</tr>
<tr>
<td>GetBlockbyShape(X1)</td>
<td>9</td>
<td>X1 block; the X1 one</td>
<td>X1: ‘shape’, O1: ‘block_set’</td>
<td>square block; the round one</td>
</tr>
<tr>
<td>GetBlockbyName(X1)</td>
<td>10</td>
<td>X1; block marked X1</td>
<td>X1: ‘name’, O1: ‘block_set’</td>
<td>A. block marked B</td>
</tr>
<tr>
<td>GetBlockAtLocation(X1,X1)</td>
<td>11</td>
<td>block at X1</td>
<td>X1: ‘location’, O1: ‘block_set’</td>
<td>block at row 2 and column 3</td>
</tr>
<tr>
<td>GetLocation(X1, X2)</td>
<td>12</td>
<td>X1 of X2</td>
<td>X1: ‘location’, X2: ‘block_set’, O1: ‘location’</td>
<td>left of blue block; below block B</td>
</tr>
</tbody>
</table>

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).

### 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’, ‘orange’, ‘blue’, ‘yellow’ 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 (R)** [Line 1, Algorithm 1]:
Given the user command $C$, $R$ rephrases $C$ 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 $R$. In this process, $R$ first reads the SC specification and forms a tagset by enumerating all argument types in the action SCs and also, builds a rephraser 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’) as the corresponding dictionary values 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., $C'$ in Figure 1 [where, $X1$="circular", $X2$="left", $X3$="$D"].

(ii) **SC Matcher [or simply Matcher (M)]**: Given the rephrased and tagged command $C'$ and the set $T$ of (action or utility) SCs, Matcher $M$ computes a match score $f(t, C')$ for each $t \in T$ and returns the top ranked SC $\hat{t} = \arg \max_{t \in T} f(t, C')$. Any paraphrasing model can be used as $M$. This work uses information retrieval (IR) based unsupervised matching models for $M$ (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 $T$ (SC store) such that for any $a \in A_{set}$, $C'$ and $a$ has an identical set of variables and types. If $A_{set} \neq \emptyset$, $M$ returns the top ranked action SC $a_0$ [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 $M$ 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 $T$ 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 ungroundability (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 to avoid error in matching due to different variable names. E.g., variable X3 in sub-expression “name/X3” [see C1 in Figure 1] is renamed as X1 [i.e. “name/X1”] (not shown), so that it can match with AID 10, while reducing C1 to C2.

- If U_set = ∅, Matcher M cannot perform reduction of C’ for each and only j is incremented [Line 18]. Otherwise, M returns the top ranked utility SC u_r for each [Line 12]. Next, C’ is replaced by reducing 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’2 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].

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 XI: ‘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 = ⟨type(X1_u), ..., type(X N_u)⟩ be the sequence of variable types in a utility SC u from left to right, where X_i_u is the i_th 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| (|·| 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 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 $R$ 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 $Q$ (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 [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 $M$) 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.

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 multi-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 ($r_{expr}, r_{para}$) pairs are used to update dictionary of $R$ [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 $R$, 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 $T$ [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 ($D$), 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 ($K_u$). For applications (1) and (2), we build $D$ and $K_u$ 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 $D$ and $K_u$ (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

---

### Algorithm 2 Interactive Knowledge Learning

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

```plaintext
1: \begin{algorithm}
2: \textbf{1:} \textbf{r}_1 \leftarrow \text{Verify_Pred_SC}(Q, C') \{r_1 \text{ is user’s response}\}
3: \textbf{2:} \textbf{if} \ r_1 = “no” \textbf{then}
4: \textbf{3:} \textbf{r}_2 \leftarrow \text{ShowSC_List}(C_{rnk}) \{C_{rnk} \text{ is the action SC rank list returned by } M\}
5: \textbf{4:} \textbf{end if}
6: \textbf{5:} \textbf{for all} variable \(x_i\) in \(r_2\) \textbf{do}
7: \textbf{6:} \(r_{expr} \leftarrow \text{Ask_Ref_Expr}(x_i, C)\)
8: \textbf{7:} \(r_{prop} \leftarrow \text{Ask_Para_Expr}(r_{expr})\)
9: \textbf{8:} \(r_{val} \leftarrow \text{Choose_Para_Val}(r_{prop}, r_{expr})\)
10: \textbf{9:} \(r_{para} \leftarrow \text{Ask_Para_Expr}(r_{val}, r_{expr})\)
11: \textbf{10:} \textbf{Update} \(R\) \textbf{with all} \((r_{val}, r_{para})\) \textbf{pairs}
12: \textbf{11:} \textbf{end for}
13: \textbf{12:} \textbf{Rephrase} \(C\) \textbf{to get a new SC and update} \(T\)
\end{algorithm}
```
written by the human users to create the test dataset (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 (K_u). Note, as CML can only ask a fixed set of questions, we can build 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 ∅ (not groundable to any of the action SCs (or APIs)). Thus, if CML returns ∅ for such commands, it is considered correct. Table 4 shows the test command dataset (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.

**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.

(1) **BERT-JISF:** This is a state-of-the-art joint intent detection and slot filling model that fine-tunes a pre-trained BERT model [8] 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 developer) 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 M. (3) **CML-vsm:** Tf-idf based vector space model is used for 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 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′(t_i). Next, we use cosine similarity as the scoring function to measure the similarity between v_c′ and v_t. We use 50D Glove [3] 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

<table>
<thead>
<tr>
<th>Application</th>
<th>0-UC</th>
<th>1-UC</th>
<th>2-UC</th>
<th>&gt;2-UC</th>
<th>UG</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WPD</td>
<td>14</td>
<td>170</td>
<td>14</td>
<td>23</td>
<td>39</td>
<td>235</td>
</tr>
<tr>
<td>FB</td>
<td>1183</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1183</td>
</tr>
</tbody>
</table>

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 ungroundable. 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.

<table>
<thead>
<tr>
<th>Models</th>
<th>BW</th>
<th>WPD</th>
<th>FB</th>
<th>BW_Acc</th>
<th>WPD_Acc</th>
<th>FB_Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-JISF</td>
<td>49.70</td>
<td>59.23</td>
<td>57.87</td>
<td>58.17</td>
<td>34.06</td>
<td>57.61</td>
</tr>
<tr>
<td>CML-jac</td>
<td>68.93</td>
<td>79.35</td>
<td>74.04</td>
<td>83.27</td>
<td>86.22</td>
<td>97.35</td>
</tr>
<tr>
<td>CML-vsm</td>
<td>68.93</td>
<td>79.35</td>
<td>74.46</td>
<td>85.81</td>
<td>88.08</td>
<td>97.35</td>
</tr>
<tr>
<td>CML-embed</td>
<td>68.63</td>
<td>79.94</td>
<td>68.93</td>
<td>82.83</td>
<td>83.43</td>
<td>97.35</td>
</tr>
<tr>
<td>CML-vsm (-R)</td>
<td>64.79</td>
<td>77.42</td>
<td>68.08</td>
<td>79.43</td>
<td>76.58</td>
<td>88.35</td>
</tr>
<tr>
<td>CML-vsm (-U)</td>
<td>14.49</td>
<td>14.49</td>
<td>11.48</td>
<td>11.70</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CML-jac + SCL</td>
<td>69.82</td>
<td>81.34</td>
<td>76.17</td>
<td>84.43</td>
<td>92.05</td>
<td>97.35</td>
</tr>
<tr>
<td>CML-vsm + SCL</td>
<td>70.11</td>
<td>80.09</td>
<td>77.02</td>
<td>85.49</td>
<td>93.32</td>
<td>97.35</td>
</tr>
<tr>
<td>CML-jac + SCL + APL</td>
<td>72.78</td>
<td>81.43</td>
<td>80.00</td>
<td>83.33</td>
<td>93.17</td>
<td>98.41</td>
</tr>
<tr>
<td>CML-vsm + SCL + APL</td>
<td>73.07</td>
<td>80.16</td>
<td>80.85</td>
<td>86.37</td>
<td>94.67</td>
<td>98.41</td>
</tr>
</tbody>
</table>

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.

---

3https://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 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 CCL 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.

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 {}) 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


