An Interactive Scribe for Guiding Task-Oriented Conversations in Enterprise Call Centers

Devin Conathan, Joseph Bockhorst, Dan Dickinson, Shailesh Acharya, Glenn Fung, Scott Rouse
(dconatha,jbockhor,ddickins,sachary1,gfung,srouse)@amfam.com
American Family Insurance
Madison, WI

ABSTRACT

We introduce an approach for guiding company representatives through customer interactions according to company protocols represented in a knowledge graph. Our proof-of-concept application features a dynamic user interface driven by a grammar derived from the knowledge graph. We present and suggest future directions for our work-in-progress demonstration which addresses many challenges that arise when building task-oriented applications at enterprise scale.

CCS CONCEPTS

• **Human-centered computing** → User interface design; • **Computing methodologies** → *Knowledge representation and reasoning*; • **Applied computing** → **Service-oriented architectures**; **Business process modeling**.

KEYWORDS

knowledge graphs, business processes, user interface design, task-oriented semantic parsing

1 INTRODUCTION

Enterprise call centers provide service to customers in the form of interactions that attempt to address customers' needs while carrying out company protocols. There is an inherent conflict here; company representatives must adhere to the strict protocols (deviations can be costly) while remaining dynamic and flexible to meet the sometimes complex and multifarious needs of the customer. Addressing this conflict without burdening the caller with a slow and stilted conversation is a challenge even for experienced representatives. To address these challenges we propose a system that serves as an interactive but unobtrusive scribe that assists the company representative to efficiently identify and navigate the appropriate protocols and accomplish their task.[1]

1.1 Background

Our system focuses on an important type of call center interaction where the task is to capture some *a priori* unknown collection of highly structured information. For example, when a customer calls an insurance company to initiate an auto claim, the company needs to determine certain details such as the cause of damage, contact information for any third parties, if there were injuries, if the vehicle is driveable and so on. Failure to capture the required information results in costs and inconveniences such as routing the claim to the wrong adjuster or not helping the customer find an appropriate repair shop and rental vehicle. Such mistakes result in inefficient claims processes and dissatisfied customers.

Often the schema of information needed to accomplish the task is not known a priori and can lead to complicated workflows which are difficult to learn and execute. In our example, a multi-vehicle collision claim requires different information than a hail damage claim (e.g. adjusters are generally not concerned with contacting third parties for hail damage claims). In some extreme cases, representatives are specialized in certain domains; however, this requires routing of incoming calls to appropriate representatives which comes with its own complexities and does not necessarily yield the best customer experience.

While technical solutions such as digital assistants and chatbots are obvious places to look for addressing the above concerns, there are considerable challenges facing designers, including:

- Company protocols frequently reside in unstructured formats, such as training websites or even paper manuals, and are thus out of reach of digital systems.
- Digitizing protocols into workflows and processes for representatives to follow is costly to create, error prone and challenging to maintain. Furthermore, even when these protocols are provided to representatives during training, they can be complex, numerous and hard to learn.
- Workflows are frequently overly restrictive by prescribing an unnecessarily rigid sequence of questions. This can lead to unnatural interactions and a poor user experience for customers and representatives.
- The speech recognition demands for speed and accuracy for a real-time digital scribe to be viable have, until very recently, exceeded the state-of-the-art.
- In settings where text is available (such as chat or when a speech recognition system has been integrated), computational parsing approaches usually fall short. While there there have been significant recent advances in machine learning and natural language processing (NLP) techniques to train statistical models for parsing utterances, such techniques are rarely applicable in specialized domains because of a dearth of labeled training data.

1.2 Proposed Solution

To address these challenges, we propose a declarative digital scribe that automatically extracts information in real time and guides representatives through phone calls according to company protocols. Our contributions include:

[1] A short video demonstration of our system is available at https://www.youtube.com/watch?v=zvId4iLoJuw

Knowledge Graphs and E-Commerce Workshop '20, August 2020, San Diego, CA, USA

- A declarative approach to storing company protocols in a knowledge graph
- A grammar derived from the knowledge graph that provides formal foundations to flexibly drive dynamic applications that require parsing and integration of fragmentary information
- A system which can (1) guide representatives through the initial claims process, (2) extract required information through incoming voice streams and (3) collect highly structured training data organized according to the graph schema and that can be used to enhance future systems

2 PRELIMINARIES

In this section, we define some terminology to help explain the theoretical underpinnings of our system. The system itself is detailed in Section 3.

2.1 Knowledge Graph

The core of our application is driven by rich data structures stored in a knowledge graph (KG). Let $\mathcal{E}$ be a set of **entities** and $\mathcal{P}$ be a set of **properties**. A **knowledge graph** $\mathcal{K} \subset \mathcal{E} \times \mathcal{P} \times \mathcal{E}$ represents a set of facts; if $(e_1, p, e_2) \in \mathcal{K}$, then entity e_1 has property p equal to e_2: (`Barack Obama`, `BIRTHPLACE`, `Honolulu`).

$\mathcal{E}$ contains both instances and types. All instances are associated with at least one type which is expressed in the graph with a special property `TYPE`. For example, the tuple (`10`, `TYPE`, `Integer`) says there's an entity `10` that's an instance of the type `Integer`. We can also have a special property `SUBTYPE` for capturing type hierarchies: (`Integer`, `SUBTYPE`, `Number`). Schemas of types can be encoded in the graph with triplets that comprise a property and two types. For example, (`Person`, `AGE`, `Integer`) indicates that instances of `Person` have an `AGE` property that is an instance of an `Integer`.[2]

Types like `Person` that have properties are called **complex** while atomic-valued types (`Integer`, `Boolean`, etc.) are called **simple**. Furthermore, types are characterized as either **concrete** or **abstract**, where concrete types can have direct instances but abstract types cannot (an abstract type can only be instantiated by instantiating one of its concrete subtypes).

This type system allows us to store arbitrarily complex data structures and schemas in our KG. Figure 1 shows a subgraph containing an example schema of a `Claim` type and its subtypes, which is a simplified version of what drives our demo.

2.2 Episode Grammar

We define an **episode** of type `T` (or, "type `T` episode") as the process of instantiating an instance of `T`. For example, an episode of the type `Person` would involve identifying a person's name, age, etc. An episode is **complete** when enough properties have been defined to create a valid instance of `Person` according to its schema.

For a given type `T` episode we use the KG to construct a context-free grammar we call the **episode grammar** (EG). A sentence in the EG corresponds to an instance of `T`. The nonterminals of the EG comprise types and properties from the KG. Each type in the type hierarchy under the `T` contributes a nonterminal to the EG. Given non-final type $\overline{\texttt{T}}$ we create a production rule for each subtype of $\overline{\texttt{T}}$.

[2]Technically a Person object must have an `AGE` property only if `AGE` is required. In our system properties are marked as either required or optional.

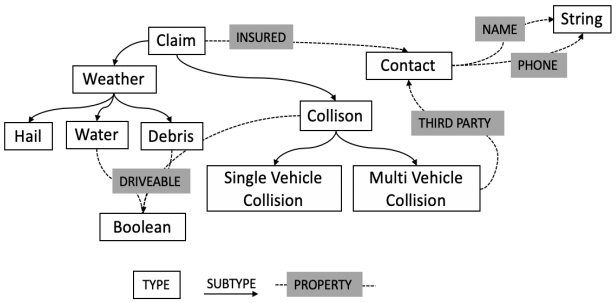

Figure 1: The subgraph describing the schema of the type `Claim`.

For example, given the schema from Figure 1, the types `Claim` and `Weather` have the following production rules:

$$\begin{aligned} \texttt{Claim} &\rightarrow \texttt{Weather} \mid \texttt{Collision} \\ \texttt{Weather} &\rightarrow \texttt{Hail} \mid \texttt{Water} \mid \texttt{Debris} \end{aligned}$$

The mapping for final type $\hat{\texttt{T}}$ with properties $p_1 \ldots p_m$ is a single production rule:

$$\hat{\texttt{T}} \rightarrow \hat{\texttt{T}}.p_1 \ldots \hat{\texttt{T}}.p_m$$

The final type `Debris`, for example, yields:

$$\begin{aligned} \texttt{Debris} &\rightarrow \texttt{Debris.INSURED} \\ &\quad\ \ \texttt{Debris.DRIVEABLE} \end{aligned}$$

Note that a type inherits properties from its supertypes. Each property $\texttt{T}.p_i$ has a single production rule $\texttt{T}.p_i \rightarrow \textsc{Type}(\texttt{T}.p_i)$. If $\textsc{Type}(\texttt{T}.p_i)$ is not in the EG it is also added using the method for adding types we just described. To continue our example, we expand the properties of `Debris`:

$$\begin{aligned} \texttt{Debris.INSURED} &\rightarrow \texttt{Contact} \\ \texttt{Contact} &\rightarrow \texttt{Contact.NAME} \\ &\quad\ \ \texttt{Contact.PHONE} \\ \texttt{Contact.NAME} &\rightarrow \texttt{String} \\ \texttt{Contact.PHONE} &\rightarrow \texttt{String} \\ \texttt{Debris.DRIVEABLE} &\rightarrow \texttt{Boolean} \end{aligned}$$

2.3 Partial Instances and the Episode Object

A type `T` episode begins by constructing a **partial instance** of `T` that we call the **episode object**. An instance is partial if either (a) its type is abstract or (b) any of its required properties are undefined or partial. Typically, an episode starts with an "empty" partial instance, whose properties are all undefined. A non-partial instance is also called **complete**. An episode is complete when the episode object is complete.

There is a close relationship between episode objects and parse trees in the episode grammar. Specifically, a partial object corresponds to an incomplete parse tree while a complete object corresponds to a full parse tree in which all leaf nodes are terminals. In the remainder of the paper we stay closer to the language of episode objects as we find it more natural given the topics at hand. More examples and details about our approach to derive a grammar from a knowledge graph can be found in [1].

2.4 Algorithm

High-level pseudocode for an episode is shown in Algorithm 1. The algorithm processes a sequence of inputs x one-by-one and returns when the episode object $\mathcal{E}$ is complete. The inputs arise from interactions between the customer, the representative and the user interface (UI) and come in a variety of forms such as voice, text and form submissions.

An input first gets processed by the PARSE function which yields a set of partial objects $\mathbf{y}$. For example, a reasonable parse for text input "My name is Bob Smith" is a single partial object of type `Contact` with `NAME`="Bob Smith" and `PHONE` undefined.

Next the ASSEMBLE routine attempts to integrate the parsed partial object into the episode object. While ASSEMBLE may be a complex routine that searches for the optimal joint merger of $\mathbf{y}$ into $\mathcal{E}$, presently we employ a greedy approach in which we process the parses $y_i \in \mathbf{y}$ individually and merge a given parse into $\mathcal{E}$ only if there is a single *valid merger*.

Let a **merger** be a pair $(y, \vec{z})$ where y is a partial object and $\vec{z} = (\mathtt{S}, p_1, \ldots, p_n)$ is a *path* that unambiguously specifies a location in the episode object. A **path** consists of exactly one type `S` optionally followed by any number of properties[3] $p_1 \ldots p_n$. For example, the corresponding path for the parse from the "Bob Smith" example above is `Claim.insured`. Within an episode of type `T`, `S` must be `T` or one of its subtypes and the properties must be consistent with the schema in the KG.

A necessary condition for the validity of merger $(y, \vec{z})$ involves type coherency between the partial object and its integration position within the episode object indicated by the path. Specifically, $\text{TYPE}(y)$ must be the same as or a subtype of $\text{TYPE}(\vec{z})$ where we define the type for a path to be the type of its terminal element. That is, $\text{TYPE}(\vec{z})$ is either `S` (when $\vec{z}$ has no properties) or $\text{TYPE}(p_n)$. Other validity conditions depend on the dialog state. For example, we prevent partial objects that arise from parses of utterances to overwrite values explicitly set via UI form inputs. Importantly, we save these valuable mistakes as they serve as highly-relevant negative examples that we can use to train (or re-train) statistical parsers to improve the performance of the system.

Algorithm 1 Pseudocode for an episode

```
function EPISODE(E, θ)
    E: the episode object
    θ: dialog state
    if ISCOMPLETE(E) then
      return E
    else
      x ← INPUT()
      y, θ ← PARSE(y, θ)
      E ← ASSEMBLE(E, θ, y)
      return EPISODE(E, θ)
    end if
end function
```

[3] Our approach may be trivially extended to support indexed collections of similarly typed objects including arrays (indexed by integers) and dictionaries (indexed by strings).

3 SYSTEM OVERVIEW

3.1 Knowledge Graph Construction

In the previous section we detailed how the knowledge graph and the types and schemas it contains determine the behavior of the system, so obviously constructing the correct knowledge graph is key. As this work is still in a proof-of-concept phase, we manually construct the knowledge graph based on reading and interpreting company protocols in their unstructured formats. In the future, we are interested in exploring using knowledge graph construction techniques to automatically generate the knowledge graph from unstructured sources.

3.2 Frontend and User Experience

The user interface (UI) is essentially a dynamic form which renders the current state of the episode object. Undefined properties manifest as input fields. Abstract types appear as buttons for the user to select. The knowledge graph contains metadata about how choices and input fields should be displayed to the user. For example, in Figure 3a, selecting the appropriate `Claim` subtype appears as buttons whereas the `Contact` object appears as text boxes for the user to fill in. The triplets (`Claim`, `QUESTION`, `"What happened?"`) and (`Name`, `QUESTION`, `"What is your name?"`) would be in our graph to indicate what should appear above the buttons or fields.

3.3 Voice Input

The backend of our system has the ability to parse text inputs as described in the next section. In order to guide a representative through a call, we have included a voice-to-text module which transcribes the incoming channels from the representative and customer which can then be passed to the backend as text. The streams of voice data are broken into utterances, and each utterance is passed to a server running the voice-to-text speech recognition model. We use Mozilla's implementation [3] of the Deep Speech architecture [2]. Once the utterance has been transcribed it is passed to the backend to be processed as a text input.

3.4 Parsing text inputs

After receiving a text input, our system parses it and produces any number of partial objects. Which parsers are applied depend on the current episode object and the current state of the dialog (which is detailed in Section 3.6). For example, at the beginning when our episode object is a `Claim` object, we would apply a parser that looks for text like "crashed" or "accident", which would produce an empty `Collision` partial object. Figure 2 shows an example where the phrase "I got rear ended" is parsed as an empty `MultiVehicleCollision` object.

The parsers are stored in the KG as special properties of types and instances so they can be retrieved easily via queries. Our demo employs simple pattern-matching parsers, but the system allows for more complicated statistical parsers. Indeed, one of the major features of our system is that we are continuously collecting training data because a human is ultimately vetting the information and submitting the final form.

3.5 Processing inputs from UI

In addition to parsing voice and unstructured text inputs, our system can also take in structured information from the UI. This feature is essential since we cannot expect our parsers to accurately extract all the necessary information for every interaction. We believe this addresses the challenges surrounding the shortcomings of statistical parsers brought forth in Section 1.1 in two ways: (1) it allows for a human-in-the-loop interaction to correct mistakes of the statistical parsers, and (2) it generates new training data to improve the accuracy of the parsers in the future.

Figure 3 shows some screenshots of our UI. Note that some inputs (e.g. `NAME`) are text fields while others (e.g. the appropriate `CLAIM` subtype) can be rendered as buttons. Metadata specifying how the fields and options should be displayed in the UI are included in the KG.

Inputs from the UI are processed similarly to text inputs. The submission of a field produces a partial object which is then assembled into the episode object. For example, filling out the `NAME` field in Figure 3a would correspond to creating a `Contact` partial object with the given name.

3.6 State Tracking

The episode object functions as a state tracker for our system; it contains all the data that we take in and is what ultimately gets submitted when the task is accomplished. Within the state we also track the origin of values to avoid overwriting values set via UI form inputs as mentioned in Section 2.4.

Some dialog state tracking is necessary in order to parse responses to questions in the conversation. For example, if we received the response "yes" from a customer, we would need to know that this was in response to the question "Is your car still driveable?" in order to successfully parse it as the appropriate partial object. To handle this situation, we apply parsers to the speech from the representative. If we recognize the question "Is your car still driveable" we would then make a "yes"/"no" parser available for the response from the customer that would map "yes" to a partial object with `DRIVEABLE` set to `True`.

3.7 Assemble and Update

Partial objects obtained through the UI or by parsing text are assembled into the episode object according to the algorithm detailed in Section 2.4. The UI is updated according to the new episode object and any relevant metadata from the KG.

3.8 Completion

When the episode is complete, the representative is given the ability to review the information before submitting the object, thus concluding the interaction. The final episode object is usually submitted to a backend processing system. Additionally, we store an enriched version of the object with the raw inputs and any corrections to be used for training data in the future.

4 CONCLUSIONS AND FUTURE WORK

We have a proposed a knowledge graph-driven system for helping company call center representatives naturally interact with customers while at the same time meeting company protocols. Our system's design facilitates the natural collection of weakly-labeled data connecting user utterances to KG entities and types. In future work we plan to investigate algorithms for system improvement via automatic and human-in-the-loop machine learning algorithms. Additional future work includes the design and implementation of an A/B testing methodology to objectively measure the impact of the system on day-to-day operations.

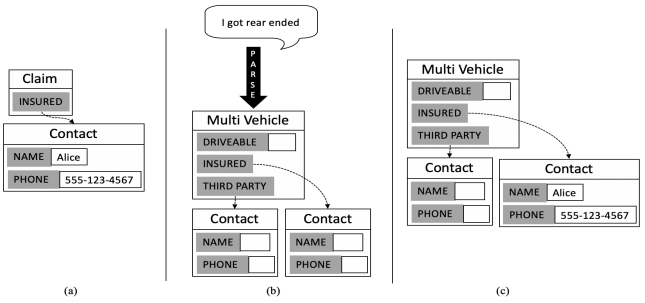

Figure 2: An example showing Algorithm 1 in practice. (a) shows the episode object before processing the input, (b) shows the parser producing a partial object from the text input, and (c) shows the updated episode object after assembly.

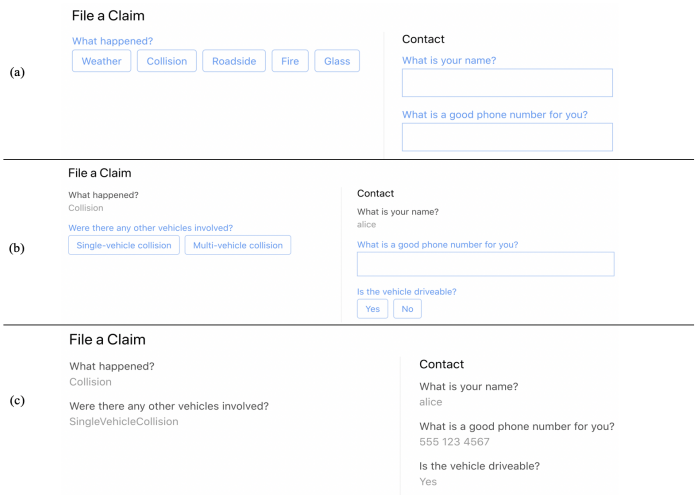

Figure 3: An example showcasing the system's dynamic UI. (a) shows the initial screen. In (b) we see that the claim has been marked as a `Collision` claim and more information is necessary, and (c) shows the claim with all required information and ready to submit.

REFERENCES

[1] Joseph Bockhorst, Devin Conathan, and Glenn Fung. Knowledge graph-driven conversational agents. *Knowledge Representation & Reasoning Meets Machine Learning (NeurIPS Workshop)*, 2019.

[2] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[3] Mozilla. A tensorflow implementation of baidu's deepspeech architecture, 2019.