An End User Development Approach for Failure Handling in Goal-oriented Conversational Agents

Toby Jia-Jun Li, Igor Labutov, Brad A. Myers, Amos Azaria, Alexander I. Rudnicky and Tom M. Mitchell

Abstract This chapter introduces an end user development (EUD) approach for handling common types of failures encountered by goal-oriented conversational agents. We start with identifying three common sources of failures in human-agent conversations: *unknown concepts*, *out-of-domain tasks* and *wrong fulfillment means or level of generalization in task execution*. To handle these failures, it is useful to enable the end user to program the agent and to "teach" the agent what to do as a fallback strategy. Showing examples for this approach, we walk through our two integrated systems: SUGILITE and LIA. SUGILITE uses the programming by demonstration (PBD) technique, allowing the user to program the agent by demonstrating new tasks or new means for completing a task using the GUIs of third-party smartphone apps, while LIA learns new tasks from verbal instructions, enabling the user to teach the agent through breaking down the procedure verbally. LIA also supports the user to verbally define unknown concepts used in the commands and adds those concepts into the agent's ontology. Both SUGILITE and LIA can generalize what they have learned from the user across related entities and perform a task with new parameters in a different context.

T.J.-J. Li · I. Labutov · B.A. Myers · A.I. Rudnicky · T.M. Mitchell
Carnegie Mellon University, Pittsburgh, PA e-mail: tobyli@cs.cmu.edu

I. Labutov
e-mail: ilabutov@cs.cmu.edu

B.A. Myers
e-mail: bam@cs.cmu.edu

A.I. Rudnicky
e-mail: air@cs.cmu.edu

T.M. Mitchell
e-mail: tom.mitchell@cs.cmu.edu

A. Azaria
Ariel University, Isarel e-mail: amos.azaria@ariel.ac.il

1 Introduction

Recently, goal-oriented conversational agents have made significant progress in expanding the support for more task domains, refining the speech recognition components for reducing recognition errors, improving the natural language understanding model for better comprehending user commands, and enhancing the dialog management system to have more natural and fluent conversations with users.

Despite the progress made, these conversational agents still inevitably encounter errors and failures. In designing and implementing the user experience of a goal-oriented conversational agent, a crucial part is *failure handling* – how should the agent respond when it could not understand the user's command, could not perform the task the user asked for, or performed the task in a different way than what the user had hoped for? Many UX guidelines for designing conversational flow and fallback strategies in dialog management systems have been proposed, such as directing the users to rephrase in a different way that the agent can understand (e.g., [2, 21]), refocusing and redirecting the users to tasks that the agent can support (e.g., [6]), updating and confirming the beliefs of the system (e.g., [7]), or providing users options to correct the input (e.g., [3]). Extensive work in dialog management systems have also been conducted to detect and to recover from errors of non-understanding [8, 9], and to proactively elicit knowledge [21] from users for helping reduce subsequent task failures in user's dialogs with conversational agents.

In this chapter, we take a different perspective to discuss an end-user development (EUD) approach for failure handling in conversational agents. We believe that the design goal for conversational agents should evolve from *easy to use* (i.e. users should easily figure out how to provide a "good" or "valid" input for the agent with the help of good UX design) to *easy to develop* [19], where the end users are empowered to "teach" the agent what to do for a given input, and to modify how a task is executed by the agent. Such new capabilities can provide expandability, customizability and flexibility to the users, catering to their highly diverse and rapidly changing individual preferences, needs, personal language vocabulary and phrasings. We share our experience on designing and implementing LIA and SUGILITE, two integrated systems that can enable end users of the conversational agent to define new concepts in the ontology through giving natural language instructions, and to instruct the agents new tasks by demonstrating using the existing third-party apps available on the phone. The two systems leverage different input modalities – LIA relies on the user instructions in natural language, while SUGILITE mainly uses the user's demonstrations on graphical user interfaces (GUI). We also discuss our proposed future work on further combing the two approaches to provide the users with multi-modal end user development experiences for conversational agents.

This chapter first explains three types of challenges that we identified on how goal-oriented conversational agents handle failures in the dialog. Then we use our two systems LIA and SUGILITE as examples for explaining how supporting end user development can be adopted as a useful failure handling strategy for a conversatinal agent. Finally, we discuss the directions of our ongoing and planned follow-up work.

2 Underlying Challenges

This section identifies three underlying challenges for goal-oriented conversational agents in handling failures.

2.1 Understanding unknown and undefined concepts

A great amount of work in the domains of speech recognition and natural language processing has been put into helping the conversational agents better understand the verbal commands. Large ontologies and knowledge graphs of both general and domain-specific knowledge have been connected to the conversational agents, so they can identify entities and concepts in the commands, understand relations, and perform reasoning on these entities, concepts and relations (e.g., [13, 14]).

However, many of the concepts involved in the commands can be fuzzy, personal or have varying meanings that are specific to individual users, so they cannot be found in existing ontologies. For example, a user may say things like "`Show my important meetings.`" or "`Send emails to my colleagues.`", where concepts like "important meetings" and "my colleagues" are not clearly defined. There might also be customized properties (e.g., each colleague has an affiliation) and criterias (e.g., colleagues all have email addresses from the university domain) for those concepts. As result, the conversational agents are not able to understand these commands. Even some agents like Apple Siri, which actively building personal ontologies through mining personal data (e.g., emails, contacts, calendar events) [14], are still unlikely to cover all the concepts that the user may refer to in the commands due to both technical difficulties and privacy concerns. It remains an important challenge for conversational agents to learn those personal concepts found in user commands that are unknown to the agent and undefined in existing ontologies [4].

In addition to understanding unknown classes of entities, a closely related problem is understanding unknown language referring to actions. For example, the user might request "`Drop a note to Brad to say I'll be late`," but the agent might now know the action "drop a note." The problem of undefined concepts is both an ontology problem, and a language understanding problem.

2.2 Executing out-of-domain tasks

Traditionally, conversational agents were only able to perform tasks on a fixed set of domains, whose fulfillments had been pre-programmed by the developers of the agents. For example, Apple Siri used to able to only perform tasks with built-in smartphone apps (e.g., Phone, Message, Calendar, Music) and query a few integrated external web services (e.g., Weather, Wikipedia, Wolfram). Consequently,

many commands that users give are out-of-domain for the agents, despite that there are apps available on the phone for executing these tasks. In these situations, the agents generally rely on fallback strategies, such as using web search which do not perform the task. Although many of existing agents have made efforts to understand some parts of the query so they can provide more specific response than the most general "Do you want me to Google this for you? ", those out-of-domain failures are still frustrating for users in most cases. For example, we can observe the following dialog when a user tries to order a cup of coffee using Apple Siri:

User: Order a cup of Cappuccino.
Siri: Here's what I found: (Showing a list of Apple Map results of nearby coffee shops)

In the above example, we can see that although the agent has successfully recognized the user's intent, it cannot fully fulfill the user's request due to the lack of API for task fulfillment.

To address this issue, major "virtual assistant" agents like Apple Siri, Google Assistant and Amazon Alexa have started opening their APIs to third-party developers so developers can develop "skills" for the agents, or integrate their own apps into the agents. However, based on what we can observe, only a small number of the most popular third-party apps have been integrated into these agents due to the cost and engineering effort required. In the forseeable future, the "long-tail" of apps are not likely to be supported by the agents. While each "long-tail" app does not have a massive user base, some of these apps are used frequently by a small group of users. Thus, it can still be important to enable the agents to invoke commands supported by these apps.

2.3 Using the right fulfillment means and generalization in task execution

Another type of failures can happen when the agent understands the command correctly, has the capability to perform the task, but executes the task in a different fashion, with different specifications, or through a different means than what the user has hoped for. This type of failure is often a result of assumptions made by the bot developers on how the user might want the task to be performed.

An example of this type of failures is on choosing the service to use for fulfilling the user's intent. Virtual assistants like Apple Siri and Google Assistant often have a pre-set service provider for each supported task domain, despite all the other apps installed on the phone that can also perform the same task. In reality, each individual user may have a different preferred service provider to use, or even a logic for choosing a service provider on the fly (e.g., choosing the cheapest ride for the destination between Uber and Lyft). As a result, the conversational agent should allow the user to customize on what service, or how to choose the service to use for fulfilling an intent.

In some other situations, the agent makes too many assumption about the task parameters and asks too few questions. For example, in the customer reviews for an Alexa skill of a popular pizza-delivery chain, a customer writes

> "*. . . You can only order pizza; no sides, no drinks, just pizza. You also can't order specials/deals or use promo codes. Lastly, you can't select an address to use. It will automatically select the default address on the account.*" [20]

This skill also does not support any customization of the pizza, only allowing the user to choose between a few fixed options. In contrast, the Android app for the same chain supports all the features mentioned in the customer's review, plus a screen for the user to fully customize the crust, the sauce and the toppings of the pizza. This phenomenon is not unique for many conversational agents, they only support a subset of functionality and features compared with their smartphone app counterparts.

On the other hand, it's also crucial to keep the dialog brief and avoid overwhelming users with too many choices and options. This is especially true in speech interfaces, as users can quickly skim over many available options presented in the GUI, but this is not possible when using voice. However, it is not easy to decide what parameters a conversational agent should present to the users, as users often have highly diverse needs and personal preferences. An important question to ask for one user may seem redundant for another. We argue that conversational agents should (1) make all the options available but also (2) personalize for each user on which questions to ask every time and which parameters for which the agent should simply save as default values.

3 Learning from Demonstration

Learning from the user's demonstration allows a conversational agent to learn how to perform a task from observing the user doing it. This approach helps the agent address the two challenges of *executing out-of-domain tasks* and *using the right fulfillment means and generalization in task execution* in failure handling as described in the previous sections. For out-of-domain tasks, the users can program these tasks by demonstration for the agent. The users can also freely customize how a task is executed by the agent and incorporate their personal preferences in the task demonstration.

In this section, we introduce our programming by demonstration agent named SUGILITE as an example to illustrate how programming by demonstration (PBD) can be adopted in failure handling for conversational agents. We also walk through the design goals of SUGILITE and how the implementation of SUGILITE can support the design goals.

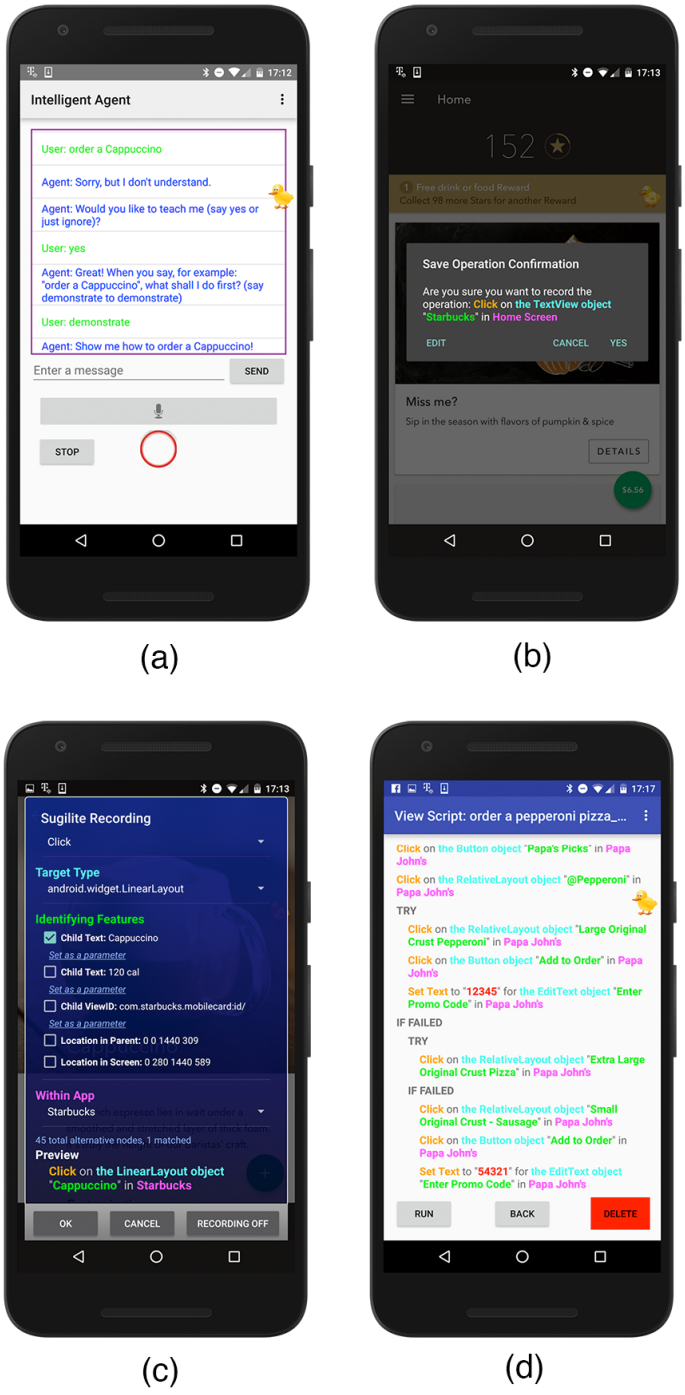

Fig. 1 Screenshots of the current version of SUGILITE prototype: (a) the conversational interface; (b) the recording confirmation popup; (c) the recording disambiguation/operation editing panel and (d) the viewing/editing script window (showing a *different* script).

3.1 SUGILITE

SUGILITE [17] is a multi-modal programming by demonstration system that enables end users to program arbitrary smartphone tasks for conversational intelligent agents by combining the demonstrations made by directly manipulating the regular graphical user interface (GUI) of smartphone apps, along with verbal instructions from the users. SUGILITE allows the users to invoke arbitrary third-party apps on an Android smartphone to perform previously-demonstrated tasks using voice commands without having to repeatedly navigate through the GUI of the app.

A major benefit of using the demonstration approach is *usability*. SUGILITE adopts the programming by demonstration technique, which significantly lowers the learning barrier for end users. By using SUGILITE, an end user of a conversational intelligent agent is able to "teach" the agent what to do for an out-of-domain command to expand the capabilities of the agent without necessarily knowing any programming language. Instead, the user can demonstrate the task using familiar GUIs of existing third-party mobile apps, leveraging their knowledge and experience on how to use those apps from the user's perspective. Unlike conventional programming languages which require the user to map what they see on the GUI to a textual representation of actions, the PBD approach enables users to program in the same environment in which they perform the actions [12].

PBD techniques have already been used in software for automating repetitive computing tasks. For example, Adobe Photoshop allows users to record a series of operations in photo editing, and to replay the recorded operations later on a different photo. Similarly, in Microsoft Office, the users can record a macro for automating operations on documents. PBD techniques have also been used in prior systems for automating tasks in domains such as email processing [11], arithmetic calculation [23], programming intelligent tutors [15] and performing web tasks [1, 16].

However, an advantage of SUGILITE compared with the prior systems is applicability. Most prior systems each focus on a single domain, while SUGILITE can be used for tasks in any arbitrary Android smartphone app, or across multiple apps. SUGILITE also doesn't require any modification of the app or access to the app's internal structures, unlike CHINLE [10] (requiring the app to be implemented in the SUPPLE framework) or COSCRIPTER [16] (leveraging the DOM structure of the web page).

3.2 An Example Scenario

In this section, we walk through an example use case of SUGILITE to exhibit how the PBD approach can help an end user to program an out-of-domain task for a goal-oriented conversational agent by demonstration. In this example, the agent fails to fulfill the user's command to order a cup of Cappuccino. To respond, the user demonstrates ordering a cup of Cappuccino using the Starbucks app. SUGILITE then generalizes the script so the agent learns to order any Starbucks drink.

Suppose a user first asks the conversational agent, "`Order a Cappuccino.`" (Figure 1a), for which the agent answers "`Sorry but I don't understand. Would you like to teach me? Say demonstrate' to demonstrate.`" In our example, the user then responds, "`Demonstrate`" and starts demonstrating the procedure of ordering a Cappuccino using the Starbucks app installed on the phone. Alternatively, the user could also choose to use any other available coffee shop app for demonstrating this task.

She first clicks on the Starbucks icon on the home screen, taps on the main menu and chooses "Order", which is exactly the same procedure as what she would have done had she been ordering manually through the Starbucks app. (Alternatively, she could also say verbal commands for each step such as "`Click on Starbucks`", etc.) After each action, a confirmation dialog from SUGILITE pops up (Figure 1b) to confirm that the action has been recorded by the SUGILITE background service.

The user proceeds through the task procedure by clicking on "MENU", "Espresso Drinks", "Cappuccinos", "Cappuccino", "Add to Order" and "View Order" in sequence, which are all the same steps that she would use to perform the task manually without SUGILITE. In this process, the user could also demonstrate choosing options such as the size and the add-ins for the coffee according to her personal preferences. These customizations will be included every time when the task is invoked by the agent, allowing the user to personalize how a task should be performed by the conversational agent.

SUGILITE pops up the confirmation dialog after each click, except for the one on "Cappuccino", where SUGILITE is confused and must ask the user to choose from two identifying features on the same button: "Cappuccino" and "120cal" (Figure 1c). When finished, the user clicks on the SUGILITE status icon and selects "End Recording".

After the demonstration, SUGILITE analyzes the recording and parameterizes the script according to the voice command and its knowledge about the UI hierarchy of the Starbucks app (details in the Generalizability section).

This parameterization allows the user to give the voice command "`Order a [DRINK].`", where `[DRINK]` can be any of the drinks listed on the menu in the Starbucks app. SUGILITE can then order the drink automatically for the user by manipulating the user interface of the Starbucks app. Alternatively, the automation can also be executed by using the SUGILITE graphical user interface (GUI), or invoked externally by a third-party app using the SUGILITE API.

3.3 Key Design Goals

In this section, we identify three key design goals for PBD in goal-oriented conversational agents: *generalizability*, *robustness*, and *multi-modality*, following by discussing how the implementation of SUGILITE helps achieve these design goals.

3.3.1 Generalizability

A PBD agent should produce more than just record-and-replay macros that are very literal (e.g., sequences of clicks and keystrokes), but learn the task at a higer level of abstraction so it can perform the task in new contexts with different parameters. To achieve this, SUGILITE uses a multi-modal approach to infer the user's intents and makes correct generalizations on what the user demonstrates. SUGILITE collects a spoken utterance from the user for the intent of the task before the demonstration, automatically identifies the parameters in the utterance, matches each parameter with an operation in the demonstration of the user, and creates a generalized script.

While recording the user's demonstration, SUGILITE compares the identifying features of the target UI elements and the arguments of the operations against the user's utterance, trying to identify possible parameters by matching the words in the command. For example, for a verbal command "`Find the flights from New York to Los Angeles.`", SUGILITE identifies "New York" and "Los Angeles" as the two parameters, if the user typed "New York" into the departure city textbox and "Los Angeles" into the destination textbox during the demonstration.

This parameterization method provides control to users over the level of personalization and abstraction in SUGILITE scripts. For example, if the user demonstrated ordering a venti Cappuccino with skim milk by saying the command "`Order a Cappuccino.`", we will discover that "Cappuccino" is a parameter, but not "venti" or "skim milk". However, if the user gave the same demonstration, but had used the command, "`Order a venti Cappuccino.`" then we would also consider the size of the coffee ("venti") to be a parameter.

For the generalization of text entry operations (e.g., typing "New York" into the departure city textbox), SUGILITE allows the use of any value for the parameters. In the checking flights example, the user can give the command "`Find the flights from [A] to [B].`" for any `[A]` and `[B]` values after demonstrating how to find the flights from New York to Los Angeles. SUGILITE will simply replace the two city names by the value of the parameters in the corresponding steps when executing the automation.

When the user chooses an option, SUGILITE also records the set of all possible alternatives to the option that the user selected. SUGILITE finds these alternatives based on the UI pattern and structure, looking for those in parallel to the target UI element. For example, suppose the user demonstrates "`Order a Cappuccino.`" in which an operation is clicking on "Cappuccino" from the "Cappuccinos" menu that has two parallel options "Cappuccino" and "Iced Cappuccino". SUGILITE will first identify "Cappuccino" as a parameter, and then add "Iced Cappuccino" to the set as an alternative value for the parameter, allowing the user to order Iced Cappuccino using the same script. By keeping this list of alternatives, SUGILITE can also differentiate tasks with similar command structure but different values. For example, the commands "`Order Iced Cappuccino.`" and "`Order cheese pizza.`" invoke different scripts, because the phrase "Iced Cappuccino" is among the alternative elements of operations in one script, while "cheese pizza" would be among the alternatives of a different script. If multiple scripts can be used to execute

a command (e.g., if the user has two scripts for ordering pizza with different apps), the user can explicitly select which script to run.

A limitation of the above method in extracting alternative elements is that it only generalizes at the leaf level of a multi-level menu structure. For example, the generalized script for "`Order a Cappuccino.`" can be generalized for ordering Iced Cappuccino, but cannot be used to order drinks like a Latte or Macchiato because they are on other branches of the Starbucks "Order" menu. Since the user did not go to those branches during the demonstration, this method could not know the existence of those options or how to reach those options in the menu tree. This is a challenge of working with third-party apps, which will not expose their internal structures to us nor can we traverse the menu structures without invoking the apps on the main UI thread.

To address this issue, we created a background tracking service that records all the clickable elements in apps and the corresponding UI path to reach each element. This service can in the background run all the time, so SUGILITE can learn about all parts of an app that the user visits. Through this mechanism, SUGILITE can construct which path to navigate through the menu structure to reach all the UI elements seen by the service. The text labels of all such elements can then be added to the sets of alternative values for the corresponding parameters in scripts. This means that SUGILITE can allow the user to order drinks that are not an immediate sibling to Cappuccino at the leaf level of the Starbucks order menu tree, despite obtaining only a single demonstration.

This method has its trade-offs. First, it brings in false positives. For example, there is a clickable node "Store Locator" in the Starbucks "Order" menu. The generalizing process will then mistakenly add "Store Locator" to the list of what the user can order. Second, running the background tracking affects the phone's performance. Third, SUGILITE cannot generalize for items that were never viewed by the user. Lastly, many participants expressed privacy concerns about allowing background tracking to store text labels from apps, since apps may dynamically generate labels containing personal data like an account number or account balance.

3.3.2 Robustness

It is important to ensure robustness of PBD agents, so the agent can reliably perform the demonstrated task in different conditions. Yet, the scripts learned from demonstration are often brittle to any changes in the app's UI, or any new situations unseen in the demonstration. Error checking and handling has been a major challenge for many PBD systems [12, 19]. SUGILITE provides error handling and checking mechanism to detect when a new situation is encountered during execution, or when the app's UI changes after an update.

When executing a script, an error occurs when the next operation in the script cannot be successfully performed. There are at least three reasons for an execution error. First, it can be that the app has been updated and the layout of the UI has been changed, so SUGILITE cannot find the object specified in the operation. Second,

it can also be that the app is currently in a different state than it was during the demonstration. For example, if a user demonstrates how to request an Uber car during normal pricing, and then uses the script to request a car during surge pricing, then an error will occur because SUGILITE does not know how to handle the popup from the Uber app for surge price confirmation. Third, the execution may also be interrupted by an external event, like a phone call or an alarm.

In SUGILITE, when an error occurs, an error handling popup will be shown, asking the user to choose between three options: keep waiting, end executing, or fix the script. The "keep waiting" option will keep SUGILITE waiting until the current operation can be performed. This option should be used in situations like prolonged waiting in the app or an interrupting phone call, where the user knows that the app will eventually return to the recorded state in the script, which SUGILITE knows how to handle. The "end executing" option will simply end the execution of the current script.

The "fix the script" option has two sub-options: "replace" and "create a fork", which allow the user to either demonstrate a procedure from the current step that will replace the corresponding part of the old script, or create a new alternative fork in the old script. The "replace" option should be used to handle permanent changes in the procedure due to an app update or an error in the previous demonstration, or if the user changes her mind about what the script should do. The "create a fork" option (Figure 1d) should be used to enable the script to deal with a new situation. The forking works similarly to the try-catch statement in programming languages, where SUGILITE will first attempt to perform the original operation, and then execute the alternative branch if the original operation fails. (Other kinds of forks, branches, and conditions are planned as future work.)

The forking mechanism can also handle new situations introduced by generalization. For example, after the user demonstrates how to check the score of the NY Giants using the Yahoo! Sports app, SUGILITE generalizes the script so it can check the score of any sports team. However, if the user gives a command "`Check the score of NY Yankees.`", everything will work properly until the last step, where SUGILITE cannot find the score because the demonstrations so far only show where to find the score on an American football team's page, which has a different layout than a baseball team's page. In this case, the user creates a new fork and demonstrates how to find the score for a baseball team.

3.3.3 Multi-modality

Another design goal of ours is to support multiple modalities in the agent to provide flexibility for users in different contexts. In SUGILITE, both creating the automation and running the automation can be performed through either the conversational interface or the GUI.

The user can start a demonstration after giving an out-of-domain verbal command, for which SUGILITE will reply "`I don't understand...Would you like to teach me?`" or the user can manually start a new demonstration us-

ing the SUGILITE GUI. When teaching a new command to SUGILITE, the user can use verbal instructions, demonstrations, or a mix of both in creating the script. Even though in most cases, demonstrating on the GUI through direct manipulation will be more efficient, we anticipate some useful scenarios for instructing by voice, like when touching on the phone is not convenient, or for users with motor impairment.

The user can also execute automations by either giving voice commands or by selecting from a list of scripts. Running an automation by voice allows the user to give a command from a distance. For scripts with parameters, the parameter values are either explicitly specified in the GUI, or inferred from the verbal command when the conversational interface is used.

During recording or executing, the user has easy access to the controls of SUGILITE through the floating duck icon (See Figure 1, where the icon is on the right edge of the screen). The floating duck icon changes the appearance to indicate the current status of SUGILITE – whether it is recording, executing, or tracking in the background. The user can start, pause or end the execution/recording as well as view the current script (Figure 1d) and the script list from the pop-up menu that appears when users tap on the duck. The GUI also enables the user to manually edit a script by deleting an operation and all the subsequent operations, or to resume recording starting from the end of the script. Selecting an operation lets the user edit it using the editing panel (Figure 1c).

The multi-modality of SUGILITE enables many useful usage scenarios in different contexts. For example, the user may automate tasks like finding nearby parking or streaming audible books by demonstrating the procedures in advance by direct manipulation. Then the user can perform those tasks by voice while driving without needing to touch the phone. A user with motor impairment can have her friends or family program the conversational agent for her common tasks so she can execute them later by voice.

3.3.4 Evaluation

To evaluate how successfully end users with various levels of programming skill can operate learning by demonstration conversational agents like SUGILITE, we ran a lab-based usability analysis of SUGILITE with 19 participants across a range of programming skills (from non-programmers to skilled programmers). 65 out of 76 (85.5%) scripts created by the participants ran and performed the intended task successfully. 8 out of the 19 (42.1%) participants succeeded in all four tasks they were given (e.g., sending emails, checking sports scores, etc.), with all participants completing at least two tasks successfully.

In the results, we found no significant difference in both task completion time and task completion rate between groups of participants with different levels of programming skill. This result suggests that by using SUGILITE, non-programmers can program new tasks for conversational agents as well as programmers. Looking at the completion time of the participants, we measured that programming a repetitive task with SUGILITE and using the agent for executing the task later is more efficient

than performing the task manually if the task will be performed for more than 3 - 6 times, depending on the task.

4 Learning from Verbal Instruction

As we have discussed earlier, another major source for failures in the user's dialog with conversational agents is when the user uses unknown or undefined concepts in the command. For example, suppose a user gives the following commands to a conversational agent for creating task automation rules:

```
If there is an important email, forward it to my project
team.
Whenever there is a late homework, forward it to the TAs.
If the weather is bad in the morning, wake me up at 7 AM.
When I get an important meeting request, put it on my calendar.
```

The programming by demonstration approach we have described in the previous section allows the user to handle failures where the agent does not know how to fulfill the actions in the command, such as "forward to" and "put it on my calendar". However, the above commands also introduce another type of challenge. In the commands, the user refers to concepts such as *important email*, *my project team* and *bad weather*. For the agent, those concepts may be either undefined, or unclearly defined. Consequently, the agent might not be able to execute these commands successfully.

Besides learning from demonstration, another useful EUD strategy that can help the conversational agent to address this issue is *learning from verbal instruction* [5]. We designed a conversational task automation agent named LIA that allows the user to verbally define concepts used in the conditions and actions found in the user's verbal commands. The agent can learn those new concepts by adding them into the ontology. LIA also enables the user to teach the agent to perform a new task by verbally breaking down an unknown action into smaller steps, which complements the learning from demonstration approach used in SUGILITE especially in situations where instructing verbally is more convenient, or more natural for the user than demonstrating on the GUI.

LIA allows the user to define "trigger-action" style automation rules in natural language like the example ones shown. The popularity of tools like IFFFT (If-This-Then-That) and Zapier, which allow users to define simple trigger-action rules in visual programming environments, has demonstrated the general need for intuitive-to-use automation tools to simplify day-to-day tasks. However, we argue that without the capacity to teach such rules in natural language, such tools are fundamentally limited. This limitation stems from the fact that for these tools to be effective, the user's need must be matched to pre-built IF and THEN components in a library of functions. However, if the user's need strays beyond common scenarios (e.g., there

probably will not be an “important email” or “late homework” function in the library), these tools become difficult or impossible to use. The experience of building anything remotely more complex with the current tools (e.g., rules with multiple conditions) starts to resemble traditional programming and thus becomes inaccessible for many end users.

4.1 Learning Rules

We believe that giving a conversational agent the capacity to be taught rules and concepts verbally opens the possibility of letting users teach more complex and more personalized rules beyond what is currently possible with systems like IFFFT. The key advantage of a conversational setting is that the agent can take initiative when certain things are left ambiguous by the user — an issue which cannot be handled in more traditional programming environments (which includes visual programming tools like IFFFT and Zapier). Consider this example:

User: When I get an important meeting request, forward it to the assistant.

When the agent encounters this request, it first attempts to ground it to a known function, e.g., creating a new calendar event or sending an email. In this case, however, the Natural Language Understanding (NLU) system detects that it is not able to parse this expression into any a known function, and in that case it would ask the user if she wants to teach this as a new rule. If the user decides to teach, the agent then asks the user to break it down into steps so it can learn. A possible breakdown sequence for the above example could be:

User: Suppose I get a new email.
Agent: OK, I will wait for a new email.
User: Check if the subject mentions words like meet or meeting.
Agent: OK, I will check that, what should I do if that’s the case?
User: Also see if the sender’s email address is an email of a Colleague.
Agent: OK, I will check that, what should I do if that’s the case?
User: Then forward this email to the assistant.

Once the user finishes teaching, the agent can leverage this breakdown to generalize the command by identifying slots in the original request and allowing the user to give variations of the original command with different arguments:

User: When I get an important meeting request, forward it to my friends.

Unlike traditional programming environments, the conversational setting allows the user to:

1. Teach new conditions, such as important meeting requests
2. Let the user naturally define complex rules that consist of multiple conditions
3. Allow the system to take initiative in asking about the programming flow, reducing the burden on the user, e.g.,

Agent: Ok, I will check that. What should I do if that's the case?
Agent: What should I do if you are not available at that time? (when user asks to create a meeting on their calendar)

4.2 Learning Concepts

A key advantage of a conversational interface for learning new rules from the user's verbal instructions is that it allows the user to be more expressive by relying on concepts that have been taught previously, such as the concept of a "colleague" or an "assistant". This provides the flexibility to re-use these concepts as building blocks in defining future procedures and rules. If the user's assistant changes later, or if there are new members on the team, the rule will not have to be redefined. Thus, in attempting to express their need efficiently, the user can use concepts such as *variables* in a traditional programming language sense (without possibly even being aware of it).

The user can define concepts and concepts properties by saying things like:

User: Create a concept called a friend.
User: Create something called a colleague.
User: Most of my colleagues have a university affiliation.

Currently the system provides the user with the ability to define concepts (e.g., *student*, *professor*, *colleague*), its properties (e.g., *university affiliation*) and instantiate these concepts, for example:

User: Oscar is a colleague.
User: His affiliation is CMU.

This example illustrates two distinct types of concepts: concepts that are "crisp", e.g., colleague, and can be defined by enumerating instances of this concept, and those that are "fuzzy", e.g., important email or important requests, which cannot be defined but can have certain salient characteristics that the user can convey to the agent. The agent must have the capacity to learn both types of concepts from

the user via dialog. In teaching fuzzy concepts, the user is more likely to express characteristics such as:

Agent: What's an important email?
User: An important email will usually be from a colleague and might contain words like meet or meeting.
Agent: Are there any other characteristics of important emails?
User: They might also mention a date and a time.
User: Important emails will often be from a university domain

The challenge in taking advantage of these "fuzzy" statements is how to transform them into a statistical model which can predict whether a given instance is a member of a particular concept. In [22], we develop a method for training a natural language parser of such statements that uses a small amount of instances as examples (e.g., emails tagged with different concepts), in order to improve its ability to parse such statements.

5 Conclusion and Future Work

Conversation failures in human-agent conversations can be tackled from many different angles ranging from speech recognition, dialog management system architecture to the design of fallback responses. In this chapter, we have explored using an end user development (EUD) approach as a fallback strategy, where we allow the user to teach the agent in case of failure in human-agent conversations. We first started by identifying three major challenges in handling failures in goal-oriented conversational agents:

1. Understanding unknown and undefined concepts
2. Executing out-of-domain tasks
3. Using the right fulfillment means and generalization in task execution

Then, we discussed two strategies for using EUD to address these challenges: *learning from demonstration* and *learning from verbal instruction*, using the two agents we designed – SUGILITE and LIA respectively as examples. In particular, we highlighted the design goals for these agents and how the implementations help achieve the design goals.

For the future work, our focus will be on further combining the two sources to create an agent that learns from both the demonstration and the verbal instruction simultaneously. We envision that the verbal instructions can be used to disambiguate and enhance the demonstration, allowing the agent to learn not only the user's actions, but also the user's rationale behind those actions. On the other hand, demonstrations should also help ground the user's verbal instructions, allowing the user to refer to what's shown on the screen when teaching the agent new rules and concepts.

In particular, our future work should address the following challenges:

Data Description Problem

The data description problem is long-standing in PBD [12] – when the user demonstrates interacting with a UI element on the screen, the PBD system needs to infer the user's intent by determining what feature (e.g., text, label, id, screen location, child elements, etc.) should be used for finding which item to operate on in future executions of the script when the context may be somewhat different. In a pilot study, we asked the users to narrate their actions while demonstrating. We found that the narrations are often very helpful in disambiguating the features (e.g., the users said things like "`Click on the first item in the list.`", "`Click on the red button at the bottom of the screen.`" and "`Choose the option with the lowest price.`").

In some situations, the data description for the UI element in an action can be non-trivial. For example, one may say "`Choose the winning team.`" in a sports app, which can translate to "`Choose the name of a team that has a number next to it that is greater than the other number located next to the name of another team.`"

We plan to design a conversational user experience to allow the users to naturally narrate during the demonstration. Combining the narration and the demonstration, the agent should be able to extract a query that can be used for finding the UI element to operate on in future executions without requiring the user to manually select the set of features to use (e.g., Figure 1c) when the agent finds features of the UI element ambigious, or when the agent is unsure about the user's rationale for selecting a UI element to operate on.

Conditionals and Iterations

Expanding the support for conditionals in learning and representing tasks should significantly help increase the range of tasks that a conversational agent can learn. In section 4.1, we discussed the current supported "trigger-action" rules by our agent as well as how a user can break down the steps to evaluate conditionals in the trigger for the agent (e.g., what an important meeting request is). In our work [18], we also support the user in using phone notifications, app launches or events from external web services to trigger task automations. In future work, we plan to enable the user to verbally describe complex conditionals with references to contents shown on the current screen and also past screens for grounding.

We will also add support for looping and iterations. This will be useful for processing all the elements of a list in batch, for example to tap on each menu item on a screen, copy each entry and paste it into an email, or apply a filter to all images. In addition to manually adding the iteration, the conversational agent will allow users to verbally indicate that the task should be performed on all items (or the particularly desired subset) over a list, and will be able to engage with the user in a dialog to refine the resulting script.

Procedure Editing

We would like to enable users to explicitly specify different parameter values for an existing learned task. For example, the user should be able to run a modified variation of a previously demonstrated task by using utterances like "`Get my regular breakfast order, except instead of iced coffee, I want hot coffee.`" We wish to explore how the presentation of the conversation in the mixed-initiative interface can be designed to cater to this kind of procedure editing behaviors, leveraging the context of the existing scripts and GUI elements.

Acknowledgements This work was supported by Yahoo! through CMU's InMind project and by Samsung GRO 2015.

References

1. James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. Plow: A collaborative task learning agent. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1514. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.
2. Gillian Armstrong. Helping Your Baby Bot Learn To Chat Like A Grown Up Bot, August 2017.
3. Gillian Armstrong. The UX of not making your users shout at your chatbot, August 2017.
4. Amos Azaria and Jason Hong. Recommender systems with personality. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 207–210. ACM, 2016.
5. Amos Azaria, Jayant Krishnamurthy, and Tom M. Mitchell. Instructable Intelligent Personal Agent. 2016.
6. Josh Barkin. When Bots Fail At Conversation, August 2016.
7. Dan Bohus and Alexander I Rudnicky. Constructing accurate beliefs in spoken dialog systems. In *Automatic Speech Recognition and Understanding, 2005 IEEE Workshop on*, pages 272–277. IEEE, 2005.
8. Dan Bohus and Alexander I. Rudnicky. Error handling in the RavenClaw dialog management framework. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 225–232. Association for Computational Linguistics, 2005.
9. Dan Bohus and Alexander I. Rudnicky. Sorry, I didn't catch that!-An investigation of non-understanding errors and recovery strategies. In *6th SIGdial Workshop on Discourse and Dialogue*, 2005.
10. Jiun-Hung Chen and Daniel S. Weld. Recovering from Errors During Programming by Demonstration. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, pages 159–168, New York, NY, USA, 2008. ACM.
11. Allen Cypher. Eager: Programming repetitive tasks by demonstration. In *Watch what I do*, pages 205–217. MIT Press, 1993.
12. Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.
13. Myroslava O. Dzikovska, James F. Allen, and Mary D. Swift. Integrating linguistic and domain knowledge for spoken dialogue systems in multiple domains. In *Proc. of IJCAI-03 Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, 2003.

14. Thomas Robert Gruber, Adam John Cheyer, Dag Kittlaus, Didier Rene Guzzoni, Christopher Dean Brigham, Richard Donald Giuli, Marcello Bastea-Forte, and Harry Joseph Saddler. Intelligent automated assistant, January 17 2017. US Patent 9,548,050.
15. Kenneth R Koedinger, Vincent Aleven, Neil Heffernan, Bruce McLaren, and Matthew Hockenberry. Opening the door to non-programmers: Authoring intelligent tutor behavior by demonstration. In *Intelligent tutoring systems*, volume 3220, pages 7–10. Springer, 2004.
16. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.
17. Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, Denver, CO, 2017. ACM.
18. Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A. Myers. Programming IoT Devices by Demonstration Using Mobile Apps. In *Proceedings of the International Symposium on End User Development (IS-EUD 2017)*, Eindhoven, the Netherlands, 2017. Springer.
19. Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
20. Mitman93. Amazon.com: Customer review for Pizza Hut: Alexa Skills, 2017.
21. Aasish K. Pappu and Alexander I. Rudnicky. Knowledge acquisition strategies for goal-oriented dialog systems. In *Proceedings of SIGDIAL 2014*, SIGDIAL 2014, 2014.
22. Shashank Srivastava, Igor Labutov, and Tom Mitchell. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1528–1537, 2017.
23. Ian H. Witten. A predictive calculator. In *Watch what I do*, pages 67–76. MIT Press, 1993.