Demonstration + Natural Language: Multimodal Interfaces for GUI-Based Interactive Task Learning Agents

Toby Jia-Jun Li, Tom M. Mitchell, and Brad A. Myers

Abstract We summarize our past five years of work on designing, building, and studying SUGILITE, an interactive task learning agent that can learn new tasks and relevant associated concepts interactively from the user's natural language instructions and demonstrations leveraging the graphical user interfaces (GUIs) of third-party mobile apps. Through its multi-modal and mixed-initiative approaches for Human-AI interaction, SUGILITE made important contributions in improving the usability, applicability, generalizability, flexibility, robustness, and shareability of interactive task learning agents. SUGILITE also represents a new human-AI interaction paradigm for interactive task learning, where it uses existing app GUIs as a *medium* for users to communicate their intents with an AI agent instead of the interfaces for users to interact with the underlying computing services. In this chapter, we describe the SUGILITE system, explain the design and implementation of its key features, and show a prototype in the form of a conversational assistant on Android.

1 Introduction

Interactive task learning (ITL) is an emerging research topic that focuses on enabling task automation agents to learn new tasks and their corresponding relevant concepts through natural interaction with human users [69]. This topic is also related to the concept of *end-user development* (EUD) for task automation [65, 115]. Work in this domain includes both physical agents (e.g., robots) that learn tasks that might involve sensing and manipulating objects in the real world [7, 28], as well as software agents

T. J.-J. Li (✉) · T. M. Mitchell · B. A. Myers
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: tobyli@cs.cmu.edu

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

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

Y. Li and O. Hilliges (eds.), *Artificial Intelligence for Human Computer Interaction: A Modern Approach*, Human–Computer Interaction Series,
https://doi.org/10.1007/978-3-030-82681-9_15

that learn how to perform tasks through software interfaces [3, 10, 68, 75]. This paper focuses on the latter category.

A particularly useful application of ITL is on conversational virtual assistants (e.g., Apple Siri, Google Assistant) running on mobile phones. With the widespread popularity of mobile apps, users are utilizing them to complete a wide variety of tasks [27, 150]. These apps interact with users through graphical user interfaces (GUIs), where users usually provide inputs by direct manipulation, and read outputs from the GUI display. Most GUIs are designed with usability in mind, providing non-expert users low learning barriers to commonly-used computing tasks. App GUIs also often follow certain design patterns that are familiar to users, which helps them easily navigate around GUI structures to locate the desired functionalities [2, 38, 130].

However, GUI-based mobile apps have several limitations. First, performing tasks on GUIs can be tedious. For example, the current version of the Starbucks app on Android requires 14 taps to order a cup of venti Iced Cappuccino with skim milk, and even more if the user does not have the account information stored. For such tasks, users would often like to have them automated [6, 105, 127]. Second, direct manipulation of GUIs is often not feasible or convenient in some contexts. Third, many tasks require coordination among many apps. But nowadays, data often remain siloed in individual apps [29]. Lastly, while some app GUIs provide certain mechanisms of personalization (e.g., remembering and pre-filling the user's home location), they are mostly hard-coded. Users have few means of creating customized rules and specifying personalized task parameters to reflect their preferences beyond what the app developers have explicitly designed for.

Recently, intelligent agents have become popular solutions to the the limitations of GUIs. They can be activated by speech commands to perform tasks on the user's behalf [102]. This interaction style allows the user to focus on the high-level specification of the task while the agent performs the low-level actions, as opposed to the usual direct manipulation GUI in which the user must select the correct objects, execute the correct operations, and control the environment [25, 135]. Compared with traditional GUIs, intelligent agents can reduce user burden when dealing with repetitive tasks, and alleviate redundancy in cross-app tasks. The speech modality in intelligent agents can support hand-free contexts when the user is physically away from the device, cognitively occupied by other tasks (e.g., driving), or on devices with little or no screen space (e.g., wearables) [101]. The improved expressiveness in natural language also affords more flexible personalization in tasks.

Nevertheless, current prevailing intelligent agents have limited capabilities. They invoke underlying functionalities by directly calling back-end services. Therefore, agents need to be specifically programmed for each supported application and service. By default, they can only invoke built-in apps (e.g., phone, message, calendar, music) and some integrated external apps and web services (e.g., web search, weather, Wikipedia), lacking the capability of controlling arbitrary third-party apps and services. To address this problem, providers of intelligent agents, such as Apple, Google, and Amazon, have released developer kits for their agents, so that the developers of third-party apps can integrate their apps into the agents to allow the agents to invoke

these apps from user commands. However, such integration requires significant cost and engineering effort from app developers, therefore, only some of the most popular tasks in popular apps have been integrated into prevailing intelligent agents so far. The "long-tail" of tasks and apps have not been supported yet, and will likely not get supported due to the cost and effort involved.

Prior literature [143] showed that the usage of "long-tail" apps made up significant portion in user app usage. Smartphone users also have highly diverse usage patterns within apps [150] and wish to have more customizability over how agents perform their tasks [36]. Therefore, relying on third-party developers' effort to extend the capabilities of intelligent agents is not sufficient for supporting diverse user needs. It is not feasible for end users to develop for new tasks in prevailing agents on their own either, due to *(i)* their lack of technical expertise required, and *(ii)* the limited availability of openly accessible application programming interfaces (APIs) for many back-end services. Therefore, adding the support for interactive task learning from end users in intelligent agents is particularly useful.

1.1 Interactive Task Learning for Smartphone Intelligent Agents

To address this problem, We designed, implemented, and studied a new end-user programmable interactive task learning agent called SUGILITE[1] [80] Based on prior works in EUD and ITL for task automation, We identify the below key research challenges that SUGILITE seeks to address:

- **Usability:** SUGILITE should be usable for users without significant programming expertise. Some prior EUD systems (e.g., [100, 127]) require users to program in a visual programming language or a textual scripting language, which imposes a significant learning barrier and prevents users with limited programming expertise from using these systems.
- **Applicability:** SUGILITE should handle a wide range of common and long-tail tasks across different domains. Many existing EUD systems can only work with applications implemented with specific frameworks or libraries (e.g., [17, 32]), or services that provide open API access to their functionalities (e.g., [53]). This limits the applicability of those systems to a small subset of tasks.
- **Generalizability:** SUGILITE should learn generalized procedures and concepts that handle new task contexts and different task parameters without requiring users to reprogram from scratch. For example, macro recorders like [129] can record a sequence of input events (e.g., clicking on the coordinate (x, y)) and replay the same actions at a later time. But these macros are not generalizable, and will only perform the exact same action sequences but not tasks with variations

[1] SUGILITE is named after a purple gemstone, and stands for: **S**martphone **U**sers **G**enerating **I**ntelligent **L**ikeable **I**nterfaces **T**hrough **E**xamples.

or different parameters. Learning generalized procedures and concepts requires deeper understanding for the semantics of the medium of instruction, which in SUGILITE's case are the GUIs of existing mobile apps.

- **Flexibility:** SUGILITE should provide adequate expressiveness to allow users to express flexible automated rules, conditions, and other control structures that reflect their desired task intentions. The simple single trigger-action rule approach like [53, 56], while providing great usability due to its simplicity, is not sufficiently expressive for many tasks that users want to automate [140].
- **Robustness:** SUGILITE should be resilient to minor changes in target applications, and be able to recover from errors caused by previously unseen or unexpected situations with the user's help. Macro recorders such as [129] are usually brittle. Approaches with complicated programming synthesis or machine learning techniques (e.g., [98, 108]) usually lack transparency into the inference process, making it difficult for end users to recover from errors. Another aspect of robustness is to handle errors in natural language interactions [8, 101].
- **Shareability:** SUGILITE should support the sharing of learned task procedures and concepts among users. This requires SUGILITE to *(i)* have the robustness of being resilient to minor differences between different devices, and *(ii)* preserve the original end-user developer's privacy in the sharing process. As discussed in [75, 78], end users are often hesitant about sharing end-user-developed scripts due to the fear of accidentally including personal private information in shared program artifacts.

To address the challenges, SUGILITE takes a *multi-modal* interactive task learning approach, where it learns new tasks and concepts from end users interactively in two complementary modalities: *(i)* demonstrations by direct manipulation of third-party app GUIs, and *(ii)* spoken natural language instructions. This approach combines two popular EUD techniques—*programming by demonstration* (PBD) and *natural language programming*. In PBD, users teach the system a new behavior by directly demonstrating how to perform it. In natural language programming, users teach the system by verbally describing and explaining the desired behaviors using a natural language like English. Combining these two modalities allows users to take advantage of the easiest, most natural, and/or most effective modality based on the context for different parts of the programming task.

Through its multi-modal approach combining PBD and natural language programming, SUGILITE mitigates the shortcomings in each individual technique. Demonstrations are often too literal, making it hard to infer the user's higher level intentions. In other words, it often only records *what* the user did, but not *why* the user did it. Therefore, it is difficult to produce generalizable programs from demonstrations alone. On the other hand, natural language instructions can be very flexible and expressive for users to communicate their intentions and desired system behaviors. However, they are inherently ambiguous. In our approach, SUGILITE *grounds* natural language instructions to demonstration app GUIs, allowing *mutual disambiguation* [119], where demonstrations are used to disambiguate natural language inputs and vice versa.

1.2 Contributions

Our work contributes a new mixed-initiative multi-modal approach for intelligent agents to learn new task procedures and relevant concepts and a system that implements this approach. Specifically, this chapter describes the following contributions:

1. The SUGILITE system, a mobile PBD system that enables end users with no significant programming expertise to create automation scripts for arbitary tasks across any or multiple third-party mobile apps through a multi-modal interface combining demonstrations and natural language instructions [80] (Sect. 4).
2. A multi-modal mixed-initiative PBD disambiguation interface that addresses the data description problem by allowing users to verbally explain their intentions for demonstrated GUI actions through multi-turn conversations with the help of an interaction proxy overlay that guides users to focus on providing effective information [84] (Sect. 5.2).
3. A technique for grounding natural language task instructions to app GUI entities by constructing semantic relational knowledge graphs from hierarchical GUI structures, along with a formative study showing the feasibility of this technique with end users [84] (Sect. 5.2).
4. A PBD script generalization mechanism that leverages the natural language instructions, the recorded user demonstration, and the GUI hierarchical structures of third-party mobile apps to infer task parameters and their possible values from a single demonstration [80] (Sect. 5.3).
5. A top-down conversational programming framework for task automation that can learn both task procedures and the relevant concepts by allowing users to naturally start with describing the task and its conditionals at a high-level and then recursively clarify ambiguities, explain unknown concepts, and define new procedures through a mix of conversations and references to third-party app GUIs [88] (Sect. 5.4).
6. A multi-modal error handling and repairing approach for task-oriented conversational agents that helps users discover, identify the causes of, and recover from conversational breakdowns caused by natural language understanding errors using existing mobile app GUIs for grounding [82] (Sect. 5.5).
7. A new self-supervised technique for generating semantic embeddings of GUI screens and components that encode their textual content, visual design and layout patterns, and app metadata without requiring manual data annotation [87] (Sect. 5.6).

2 The Human-AI Collaboration Perspective

We argue that a key problem in the ITL process is to facilitate effective Human-AI collaboration. In the traditional view, programming is viewed as the process of transforming a user's existing mental plan into a programming language that the

computer can execute. However, in end-user ITL, this is not an accurate model. The user often starts with only a vague idea of what to do and needs an intelligent system's help to clarify their intents. We view ITL as a joint activity where the user and the agent share the same goal in a human-AI collaboration framework. In such mixed-initiative interactions, the user's goals and inputs come with uncertainty [4, 51]. The agent needs to show guesses of user goals, assist the user to provide more effective inputs, and engage in multi-turn dialogs with the user to resolve any uncertainties and ambiguities.

Significant progress has been made on this topic in recent years in both AI and HCI. Specifically on the AI side, advances in natural language processing (NLP) enable the agents to process users' instructions of task procedures, conditionals, concepts definitions, and classifiers in natural language [2, 6, 10], to ground the instructions (e.g., [12]), and to have dialog with users based on GUI-extracted task models (e.g., [11]). Reinforcement learning techniques allow the agent to more effectively explore action sequences on GUIs to complete tasks [13]. Large GUI datasets such as RICO [4] allow the analysis of GUI patterns at scale, and the construction of generalized models for extracting semantic information from GUIs.

The HCI community also has presented new study findings, design implications, and interaction designs in this domain. A key direction has been the design of multimodal interfaces that leverage both natural language instructions and GUI demonstrations [1, 7]. Prior work also explored how users naturally express their task intents [10, 15, 17] and designed new interfaces to guide the users to provide more effective inputs (e.g., [8]).

On one hand, AI-centric task flow exploration and program synthesis techniques often lack transparency for users to understand the internal process, and they provide the users with little control over the task fulfillment process to reflect their personal preferences. On the other hand, machine intelligence is desired because the users' instructions are often incomplete, vague, ambiguous, or even incorrect. Therefore, the system needs to provide adequate assistance to guide the users to provide effective inputs to express their intents, while retaining the users' agency, trust, and control of the process. While relevant design principles have been discussed in early foundational works in mixed-initiative interaction [5] and demonstrational interfaces [16], incorporating these ideas into the design and implementation of actual systems remains an important challenge.

A crucial factor in human-AI collaboration is the *medium*. SUGILITE presents a new human-AI interaction paradigm for interactive task learning, where it uses the GUIs of the existing third-party mobile apps as the medium for users to communicate their intents with an AI agent instead of the interfaces for users to interact with the underlying computing services. Among common mediums for agent task learning, app GUIs sit at a nice middle ground between (1) programming language, which can be easily processed by a computing system but imposes significant learning barriers to non-expert users; and (2) unconstrained visual demonstrations in the physical work and natural language instructions, which are natural and easy-to-use for users but infeasible for computing systems to fully understand without significant human-annotated training data and task domain restrictions given the current state-of-art

in natural language understanding, computer vision, and commonsense reasoning. In comparison, existing app GUIs cover a wide range of useful task domains for automation, encode the properties and relations of task-relevant entities, and encapsulate the flows and constraints of underlying tasks in formats that can be feasibly extracted and understood by an intelligent agent.

By sitting *between* the user and the GUIs of third-party apps, SUGILITE allows the user to teach the agent new task procedures and concepts by demonstrating them on existing third-party app GUIs *and* verbally explaining them in natural language. When executing a task, SUGILITE directly manipulates app GUIs on the user's behalf. This approach tackles the two major barriers in prevailing intelligent agents by *(i)* leveraging the available third-party app GUIs as a channel to access a large number of back-end services without requiring openly available APIs, and *(ii)* taking advantage of users' familiarity with app GUIs, so users can program the intelligent agent without having significant technical expertise by using app GUIs as the medium.

3 Related Work

3.1 Programming by Demonstration

SUGILITE uses the programming by demonstration (PBD) technique to enable end users to define concepts by referring to the content of GUIs from third-party mobile apps, and to teach new procedures through demonstrations with those apps. PBD is a promising technique for enabling end users to automate their activities without necessarily requiring programming knowledge—It allows users to program in the same environment in which they perform the actions. This makes PBD particularly appealing to many end users, who have little knowledge of conventional programming languages, but are familiar with how to perform the tasks they wish to automate using existing app GUIs [37, 94, 114].

A key challenge for PBD is generalization [37, 71, 94]. When an user demonstrates an instance of performing a task in a specific situation, the PBD system needs to learn the task a higher level of abstraction so that it can perform similar tasks (with different parameters, configurations etc.) in new contexts. SUGILITE improved the generalization capability compared with prior similar PBD agents such as CoScripter [75], HILC [57], Sikuli [147], and VASTA [132] through its support for parameterization (Sect. 5.3), data description disambiguation (Sect. 5.2), and concept generalization (Sect. 5.4).

SUGILITE supports domain-independent PBD by task automation by using GUIs of third-party apps. Similar approaches have also been used in prior systems. For example, Assistive Macros [129] uses mobile app GUIs, CoScripter [75], d.mix [50], Vegemite [97], Ringer [13], and PLOW [3] use web interfaces, and HILC [57] and Sikuli [147] use desktop GUIs. Macro recording tools like [129] can record a sequence of input events and replay them later. These tools are too literal—they

can only replay exactly the same procedure that was demonstrated, without the ability to generalize the demonstration to perform similar tasks. They are also brittle to any UI changes in the app. Sikuli [147], VASTA [132], and HILC [57] used the visual features of GUI entities to identify the target entities for actions—while this approach has some advantages over SUGILITE's approach, such as being able to work with graphic entities without textual labels or other appropriate identifiers, the visual approach does not use the semantics of GUI entities, which also limits its generalizability.

In human–robot interaction, PBD is often used in interactive task learning where a robot learns new tasks and procedures from the user's demonstration with physical objects [7, 20, 45, 64]. The demonstrations are sometimes also accompanied by natural language instructions [112, 133] similar to SUGILITE. While many recent works have been done in enhancing computing systems' capabilities for parsing human activities (e.g., [126]), modeling human intents (e.g., [44]), representing knowledge (e.g., [149]), from visual information from the physical world, it remains a major AI challenge to recognize, interpret, represent, learn from, and reason with visual demonstrations. In comparison, SUGILITE avoids this grand challenge by using existing app GUIs as the alternative medium for task instruction, which retains the user familiarity, naturalness, and domain generality of visual demonstration but is much easier to comprehend for a computing system.

3.2 Natural Language Programming

SUGILITE uses natural language as one of the two primary modalities for end users to program task automation scripts. The idea of using natural language inputs for programming has been explored for decades [11, 18, 95, 109]. In NLP and AI communities, this approach is also known as learning by instruction [10, 33, 68, 96].

The foremost challenge in supporting natural language programming is to deal with the inherent ambiguities and vagueness in natural language [141]. To address this challenge, a prior approach was to require users to use similar expression styles that resembled conventional programming languages (e.g., [11, 77, 125]), so that the system could directly translate user instructions into code. Despite that the user instructions used in this approach seemed like natural language, it did not allow much flexibility in expressions. This approach is not adequate for end-user development, because it has a high learning barrier for users without programming expertise—users have to adapt to the system by learning new syntax, keywords, and structures.

Another approach for handling ambiguities and vagueness in natural language inputs is to seek user clarification through conversations. For example, Iris [43] asked follow-up questions and presents possible options through conversations when initial user inputs are incomplete or unclear. This approach lowered the learning barrier for end users, as it did not require them to clearly define everything up front. It also allowed users to form complex commands by combining multiple natural language instructions in conversational turns under the guidance of the system. This

multi-turn interactive approach is also known as *interactive semantic parsing* in the NLP community [145, 146]. SUGILITE adopts the use of multi-turn conversations as a key strategy in handling ambiguities and vagueness in user inputs. However, a key difference between SUGILITE and other conversational instructable agents is that SUGILITE is domain-independent. All conversational instructable agents need to resolve the user's inputs into existing concepts, procedures, and system functionalities supported by the agent, and to have natural language understanding mechanisms and training data in each task domain. Because of this constraint, existing agents often limit their supported tasks to one or a few pre-defined domains, such as data science [43], email processing [10, 136], invoking Web APIs [137], or database queries [49, 61, 76].

SUGILITE supports learning concepts and procedures from existing third-party mobile apps regardless of the task domain. Users can explain new concepts, define task conditionals, and clarify ambiguous demonstrated actions in SUGILITE by referencing relevant information shown in app GUIs. The novel semantic relational graph representation of GUIs (details in Sect. 5.2) allows SUGILITE to understand user references to GUI content without having prior knowledge on the specific task domain. This approach enables SUGILITE to support a wide range of tasks from diverse domains, as long as the corresponding mobile apps are available. This approach also has a low learning barrier because end users are already familiar with the functionalities of mobile apps and how to use them. In comparison, with prior instructable agents, users are often unclear about what concepts, procedures, and functionalities already exist to be used as "building blocks" for developing new ones.

3.3 Multi-modal Interfaces

Multi-modal interfaces process two or more user input modes in a coordinated manner to provide users with greater expressive power, naturalness, flexibility, and portability [120]. SUGILITE combines speech and touch to enable a "speak and point" interaction style, which has been studied since early multi-modal systems like Put-that-there [23]. Prior systems such as CommandSpace [1], Speechify [60], QuickSet [121], SMARTBoard [113], and PixelTone [70] investigated multi-modal interfaces that can map coordinated natural language instructions and GUI gestures to system commands and actions. In programming, similar interaction styles have also been used for controlling robots (e.g., [55, 104]). But the use of these systems are limited to specific first-party apps and task domains, in contrast to SUGILITE which aims to be general-purpose.

When SUGILITE addresses the data description problem (details in Sect. 5.2), demonstration is the primary modality; verbal instructions are used for disambiguating demonstrated actions. A key pattern used in SUGILITE is *mutual disambiguation* [119]. When the user demonstrates an action on the GUI with a simultaneous verbal instruction, our system can reliably detect what the user did and on which UI object the user performed the action. The demonstration alone, however, does not

explain why the user performed the action, and any inferences on the user's intent would be fundamentally unreliable. Similarly, from verbal instructions alone, the system may learn about the user's intent, but grounding it onto a specific action may be difficult due to the inherent ambiguity in natural language. SUGILITE utilizes these complementary inputs to infer robust and generalizable scripts that can accurately represent user intentions in PBD. A similar multi-modal approach has been used for handling ambiguities in recognition-based interfaces [103], such as correcting speech recognition errors [138] and assisting the recognition of pen-based handwriting [67]. The recent DoThisHere [144] system uses a similar multi-modal interface for cross-app data query and transfer between multiple mobile apps.

In the parameterization and concept teaching components of SUGILITE, the natural language instructions come first. During the parametrization, the user first verbally describes the task, and then demonstrates the task from which SUGILITE infers parameters in the initial verbal instruction, and the corresponding possible values. In concept teaching, the user starts with describing an automation rule at a high-level in natural language, and then recursively defines any ambiguous or vague concepts by referring to app GUIs. SUGILITE's approach builds upon prior work like PLOW [3], which uses user verbal instructions to hint possible parameters, to further explore how GUI and GUI-based demonstrations can help enhance natural language inputs.

3.4 Understanding App Interfaces

A unique challenge for SUGILITE is to support multi-modal PBD on arbitrary third-party mobile app GUIs. Some of such GUIs can be complicated, with hundreds of entities, each with many different properties, semantic meanings, and relations with other entities. Moreover, third-party mobile apps only expose the low-level hierarchical representations of their GUIs at the presentation layer, without revealing information about internal program logic.

There has been some prior work on inferring semantics and task knowledge from GUIs. Prefab [40–42] introduces pixel-based methods to model interactive widgets and interface hierarchies in GUIs, and allowed runtime modifications of widget behaviors. Waken [12] also uses a computer vision approach to recognize GUI components and activities from screen captured videos. StateLens [48] and KITE [89] look at the sequence of GUI screens of completing a task, from which they can infer the task flow model with multiple different branches and states. The interaction mining approach used in ERICO [39] and RICO [38] captures the static (UI layout, visual features) and dynamic (user flows) parts of an app's design from a large corpus of user interaction traces with mobile apps. A similar approach was also used to learn the design semantics of mobile apps [99]. These approaches use a smaller number of discrete types of flows, GUI elements, and entities to represent GUI screens and their components, while our `Screen2Vec` uses continuous embedding in a vector space for screen representation.

Some prior techniques specifically focus on the visual aspect of GUIs. The RICO dataset [38] shows that it is feasible to train a GUI layout embedding with a large screen corpus, and retrieve screens with similar layouts using such embeddings. Chen et al.'s work [31] and Li et al.'s work [91] show that trained machine learning models can generate semantically meaningful natural language descriptions for GUI components based on their visual appearances and hierarchies. Compared with them, the `Screen2Vec` method (Sect. 5.6) used in SUGILITE provides a more holistic representation of GUI screens by encoding textual content, GUI component class types, and app-specific metadata in addition to the visual layout.

Another category of work in this area focuses on predicting GUI actions for completing a task objective. Pasupat et al.'s work [122] maps the user's natural language commands to target elements on web GUIs. Li et al.'s work [90] goes a step further by generating sequences of actions based on natural language commands. These works use the supervised approach that require a large amount of manually-annotated training data, which limits its utilization. In comparison, the `Screen2Vec` method used in SUGILITE uses a self-supervised approach that does not require any manual data annotation of user intents and tasks. `Screen2Vec` also does not need any annotation on the GUI screens themselves, unlike [148] which requires additional developer annotations for the metadata of GUI components.

SUGILITE faces a unique challenge—in SUGILITE, the user talks about the underlying task of an app in natural language while making references to the app's GUI. The system needs to have sufficient understanding about the content of the app GUI to be able to handle these verbal instructions to learn the task. Therefore, the goal of SUGILITE in understanding app interfaces is to abstract the semantics of GUIs from their platform-specific implementations, while being sufficiently aligned with the semantics of users' natural language instructions, so that it can leverage the GUI representation to help understanding the user's instruction of the underlying task.

4 System Overview

We present the prototype of a new task automation agent named SUGILITE.[2] This prototype integrates and implements the results from several of our prior research works [80–82, 84, 86–89]. The implementation of our system is also open-sourced on GitHub.[3] This section explains how SUGILITE learns new tasks and concepts from the multi-modal interactive instructions from the users.

The user starts with speaking a command. The command can describe either an action (e.g., "check the weather") or an automation rule with a condition (e.g., "If it is hot, order a cup of Iced Cappuccino"). Suppose that the agent has no prior knowledge in any of the involved task domains, then it will recursively resolve the unknown concepts and procedures used in the command. Although it does not know

[2] A demo video is available at https://www.youtube.com/watch?v=tdHEk-GeaqE.

[3] https://github.com/tobyli/Sugilite_development.

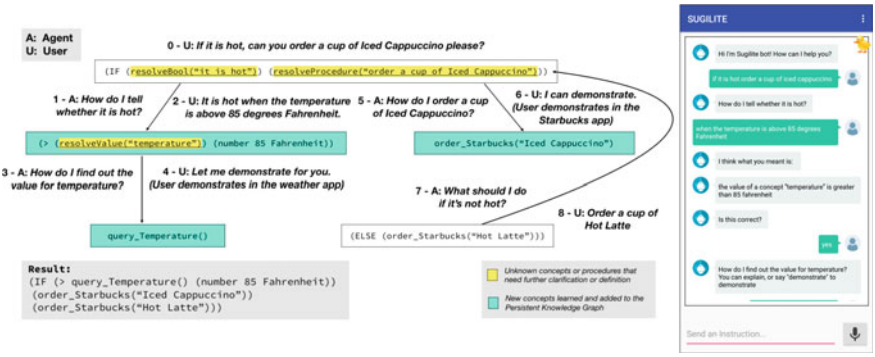

Fig. 1 An example dialog structure while SUGILITE learns a new task that contains a conditional and new concepts. The numbers indicate the sequence of the utterances. The screenshot on the right shows the conversational interface during these steps

these concepts, it can recognize the structure of the command (e.g., conditional), and parse each part of the command into the corresponding typed `resolve` functions, as shown in Fig. 1. SUGILITE uses a grammar-based executable semantic parsing architecture [92]; therefore, its conversation flow operates on the recursive execution of the `resolve` functions. Since the `resolve` functions are typed, the agent can generate prompts based on their types (e.g., "How do I tell whether…" for `resolveBool` and "How do I find out the value for…" for `resolveValue`).

When the SUGILITE agent reaches the `resolve` function for a value query or a procedure, it asks the users if they can demonstrate them. The users can then demonstrate how they would normally look up the value, or perform the procedure manually with existing mobile apps on the phone by direct manipulation (Fig. 3a). For any ambiguous demonstrated action, the user verbally explains the intent behind the action through multi-turn conversations with the help from an interaction proxy overlay that guides the user to focus on providing more effective input (see Fig. 3, more details in Sect. 5.2). When the user demonstrates a value query (e.g., finding out the value of the temperature), SUGILITE highlights the GUI elements showing values with the compatible types (see Fig. 2) to assist the user in finding the appropriate GUI element during the demonstration.

All user-instructed value concepts, Boolean concepts, and procedures automatically get generalized by SUGILITE. The procedures are parameterized so that they can be reused with different parameter values in the future. For example, for Utterance 8 in Fig. 1, the user does not need to demonstrate again since the system can invoke the newly-learned `order_Starbucks` function with a different parameter value (details in Sect. 5.3). The learned concepts and value queries are also generalized so that the system recognizes the different definitions of concepts like "hot" and value queries like "temperature" in different contexts (details in Sect. 5.4).

Fig. 2 The user teaches the value concept "commute time" by demonstrating querying the value in Google Maps. SUGILITE highlights all the duration values on the Google Maps GUI

5 Key Features

5.1 Using Demonstrations in Natural Language Instructions

SUGILITE allows users to use demonstrations to teach the agent any unknown procedures and concepts in their natural language instructions. As discussed earlier, a major challenge in ITL is that understanding natural language instructions and carrying out the tasks accordingly require having knowledge in the specific task domains. Our use of programming by demonstration (PBD) is an effective way to address this "out-of-domain" problem in both the task fulfillment and the natural language understanding processes [85]. In SUGILITE, procedural actions are represented as sequences of GUI operations, and declarative concepts can be represented as references to GUI content. This approach supports ITL for a wide range of tasks—virtually anything that can be performed with one or more existing third-party mobile apps.

Our prior study [88] also found that the availability of app GUI references can result in end users providing clearer natural language commands. In one study where we asked participants to instruct an intelligent agent to complete everyday computing tasks in natural language, the participants who saw screenshots of relevant apps used *fewer* unclear, vague, or ambiguous concepts in their verbal instructions than those who did not see the screenshots. By using demonstrations in natural language instructions, our multi-modal approach also makes understanding the user's natural language instructions easier by naturally constraining the user's expressions.

5.2 *Spoken Intent Clarification for Demonstrated Actions*

A major limitation of demonstrations is that they are too literal, and are, therefore, brittle to any changes in the task context. They encapsulate *what* the user did, but not *why* the user did it. When the context changes, the agent often may not know what to do, due to this lack of understanding of the user intents behind their demonstrated actions. This is known as the *data description problem* in the PBD community, and it is regarded as a key problem in PBD research [37, 94]. For example, just looking at the action shown in Fig. 3a, one cannot tell if the user meant "the restaurant with the most reviews", "the promoted restaurant", "the restaurant with 1,000 bonus points", "the cheapest Steakhouse", or any other criteria, so the system cannot generate a description for this action that accurately reflects the user's intent. A prior approach is to ask for multiple examples from the users [106], but this is often not feasible due to the user's inability to come up with useful and complete examples, and the amount of examples required for complex tasks [74, 116].

SUGILITE's approach is to ask users to verbally explain their intent for the demonstrated actions using speech. Our formative study [84] with 45 participants found that end users were able to provide useful and generalizable explanations for the intents of demonstrated actions. They also commonly used in their utterances semantic references to GUI content (e.g., "the close by restaurant" for an entry showing the text "596 ft") and implicit spatial references (e.g., "the score for Lakers" for a text object that contains a numeric value and is right-aligned to another text object "Lakers").

Based on these findings, we designed and implemented a multi-modal mixed-initiative intent clarification mechanism for demonstrated actions. As shown in Fig. 3, the user describes their intention in natural language, and iteratively refines the descriptions to remove ambiguity with the help of an interactive overlay (Fig. 3d). The overlay highlights the result from executing the current data description query, and helps the user focus on explaining the key differences between the target object (highlighted in red) and the false positives (highlighted in yellow) of the query.

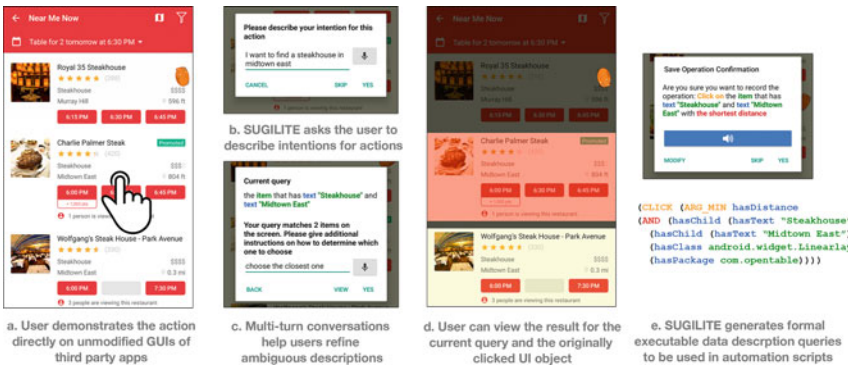

Fig. 3 The screenshots of SUGILITE's demonstration mechanism and its multi-modal mixed-initiative intent clarification process for the demonstrated actions

To ground the user's natural language explanations about GUI elements, SUGILITE represents each GUI screen as a *UI snapshot graph*. This graph captures the GUI elements' text labels, meta-information (including screen position, type, and package name), and the spatial (e.g., `nextTo`), hierarchical (e.g., `hasChild`), and semantic relations (e.g., `containsPrice`) among them (Fig. 4). A semantic parser translates the user's explanation into a graph query on the UI snapshot graph, executes it on the graph, and verifies if the result matches the correct entity that the user originally demonstrated. The goal of this process is to generate a query that uniquely matches the target UI element and also reflects the user's underlying intent.

5.2.1 UI Snapshot Graph

Formally, we define a UI snapshot graph as a collection of *subject-predicate-object triples* denoted as (s, p, o), where the subject s and the object o are two entities, and the predicate p is a directed edge representing a relation between the subject and the object. In APPINITE's graph, an entity can either represent a view in the GUI, or a typed (e.g., string, integer, Boolean) constant value. This denotation is highly flexible—it can support a wide range of nested, aggregated, or composite queries. Furthermore, a similar representation is used in general-purpose knowledge bases such as DBpedia [9], Freebase [22], Wikidata [142], and WikiBrain [83], which can enable us to plug APPINITE's UI snapshot graph into these knowledge bases to support better semantic understanding of app GUIs in the future.

The first step in constructing a UI snapshot graph from the hierarchical tree extracted from the Android Accessibility Service is to flatten all views in the tree into a collection of view entities, allowing more flexible queries on the relations between entities on the graph. The hierarchical relations are still preserved in the graph, but converted into `hasChild` and `hasParent` relationships between the corresponding view entities. Properties (e.g., coordinates, text labels, class names) are also converted into relations, where the values of the properties are represented as entities. Two or more constants with the same value (e.g., two views with the same class name) are consolidated as a single constant entity connected to multiple view entities, allowing easy querying for views with shared properties values.

In GUI designs, horizontal or vertical alignments between objects often suggest a semantic relationship [5]. Generally, smaller geometric distance between two objects also correlates with higher semantic relatedness between them [46]. Therefore, it is important to support spatial relations in data descriptions. APPINITE adds spatial relationships between view entities to UI snapshot graphs based on the absolute coordinates of their bounding boxes, including `above`, `below`, `rightTo`, `leftTo`, `nextTo`, and `near` relations. These relations capture not only explicit spatial references in natural language (e.g., the button next to something), but also implicit ones (see Fig. 4 for an example). In APPINITE, thresholds in the heuristics for determining these spatial relations are relative to the dimension of the screen, which supports generalization across phones with different resolutions and screen sizes.

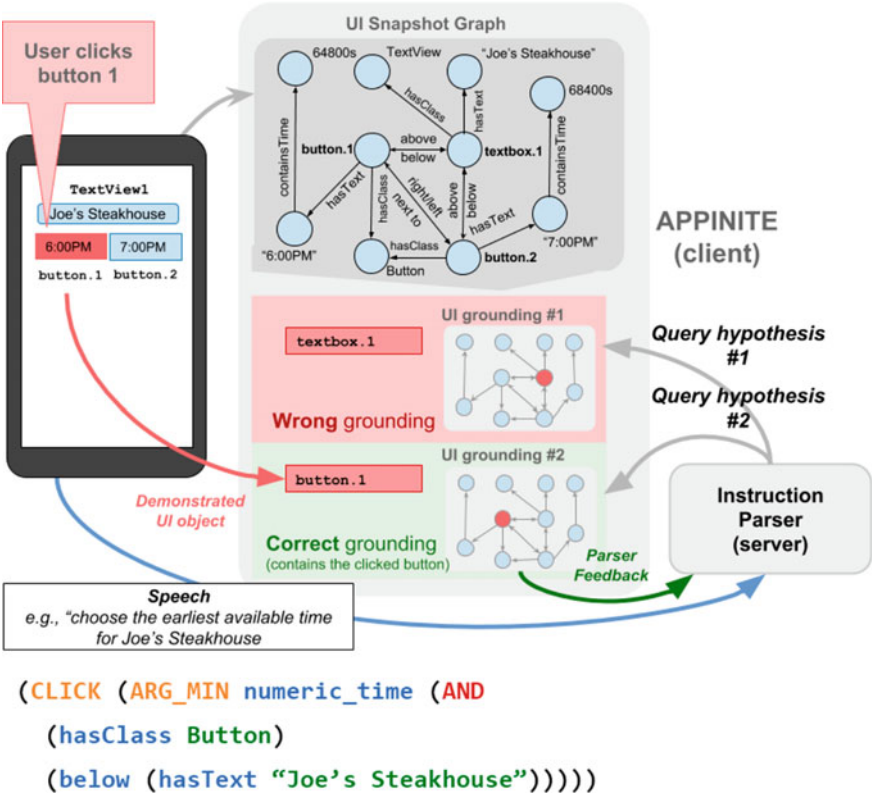

Fig. 4 SUGILITE's instruction parsing and grounding process for intent clarifications illustrated on an example UI snapshot graph constructed from a simplified GUI snippet

APPINITE also recognizes some semantic information from the raw strings found in the GUI to support grounding the user's high-level linguistic inputs (e.g., "*item with the lowest price*"). To achieve this, APPINITE applies a pipeline of data extractors on each string entity in the graph to extract structured data (e.g., phone number, email address) and numerical measurements (e.g., price, distance, time, duration), and saves them as new entities in the graph. These new entities are connected to the original string entities by `contains` relations (e.g., `containsPrice`). Values in each category of measurements are normalized to the same units so they can be directly compared, allowing flexible computation, filtering, and aggregation.

5.2.2 Parsing

Our semantic parser uses a Floating Parser architecture [123] and is implemented with the SEMPRE framework [16]. We represent UI snapshot graph queries in a

simple but flexible LISP-like query language (*S*-expressions) that can represent joins, conjunctions, superlatives and their compositions, constructed by the following 7 grammar rules:

$$E \rightarrow e;\ E \rightarrow S;\ S \rightarrow (\texttt{join}\ r\ E);\ S \rightarrow (\texttt{and}\ S\ S)$$
$$T \rightarrow (\texttt{ARG\_MAX}\ r\ S);\ T \rightarrow (\texttt{ARG\_MIN}\ r\ S);\ Q \rightarrow S \mid T$$

where Q is the root non-terminal of the query expression, e is a terminal that represents a UI object entity, r is a terminal that represents a relation, and the rest of the non-terminals are used for intermediate derivations. SUGILITE's language forms a subset of a more general formalism known as Lambda Dependency-based Compositional Semantics [93], which is a notationally simpler alternative to lambda calculus which is particularly well-suited for expressing queries over knowledge graphs. More technical details and the user evaluation are discussed in [84].

5.3 Task Parameterization Through GUI Grounding

Another way SUGILITE leverages GUI groundings in the natural language instructions is to infer task parameters and their possible values. This allows the agent to learn generalized procedures (e.g., to order *any kind of beverage* from Starbucks) from a demonstration of a specific instance of the task (e.g., ordering an iced cappuccino).

SUGILITE achieves this by comparing the user utterance (e.g., "order a cup of iced cappuccino") against the *data descriptions* of the target UI elements (e.g., click on the menu item that has the text "Iced Cappuccino") and the arguments (e.g., put "Iced Cappuccino" into a search box) of the demonstrated actions for matches. This process grounds different parts in the utterances to specific actions in the demonstrated procedure. It then analyzes the hierarchical structure of GUI at the time of demonstration, and looks for alternative GUI elements that are in parallel to the original target GUI element structurally. In this way, it extracts the other possible values for the identified parameter, such as the names of all the other drinks displayed in the same menu as "Iced Cappuccino"

The extracted sets of possible parameter values are also used for disambiguating the procedures to invoke, such as invoking the `order_Starbucks` procedure for the command "order a cup of *latte*", but invoking the `order_PapaJohns` procedure for the command "order a *cheese pizza*."

5.4 Generalizing the Learned Concepts

In addition to the procedures, SUGILITE also automatically generalizes the learned *concepts* in order to reuse parts of existing concepts as much as possible to avoid requiring users to perform redundant demonstrations [88].

For Boolean concepts, SUGILITE assumes that the type of the Boolean operation and the types of the arguments stay the same, but the arguments themselves may differ. For example, for the concept "hot" in Fig. 1, it should still mean that a temperature (of something) is greater than another temperature. But the two in comparison can be different constants, or from different value queries. For example, suppose after the interactions in Fig. 1, the user instructs a new rule "*if the oven is hot, start the cook timer.*" PUMICE can recognize that "hot" is a concept that has been instructed before in a different context, so it asks "*I already know how to tell whether it is hot when determining whether to order a cup of Iced Cappuccino. Is it the same here when determining whether to start the cook timer?*" After responding "No", the user can instruct how to find out the temperature of the oven, and the new threshold value for the condition "hot" either by instructing a new value concept, or using a constant value.

The generalization mechanism for value concepts works similarly. PUMICE supports value concepts that share the same name to have different query implementations for different task contexts. For example, following the "if the oven is hot, start the cook timer" example, suppose the user defines "hot" for this new context as "*The temperature is above 400 degrees.*" PUMICE realizes that there is already a value concept named "temperature", so it will ask "*I already know how to find out the value for temperature using the Weather app. Should I use that for determining whether the oven is hot?*", to which the user can say "No" and then demonstrate querying the temperature of the oven using the corresponding app (assuming the user has a smart oven with an in-app display of its temperature).

This mechanism allows learned concepts like "hot" to be reused at three different levels: *(i)* exactly the same (e.g., the temperature of the weather is greater than 85°F); *(ii)* with a different threshold (e.g., the temperature of the weather is greater than x); and *(iii)* with a different value query (e.g., the temperature of *something else* is greater than x).

5.5 Breakdown Repairs in Task-Oriented Dialogs

Another important challenge in facilitating effective human-AI collaboration with ITL agents is to support the discovery and repair of conversational breakdowns. Despite the advances in the agent's natural language understanding capabilities, it is still far from being able to understand the wide range of flexible user utterances and engage in complex dialog flows [47]. Existing agents employ rigid communication patterns, requiring that users adapt their communication patterns to the needs of the system instead of the other way around [14, 59]. As a result,conversational breakdowns, defined as failures of the system to correctly understand the intended meaning of the user's communication, often occur. Conversational breakdowns decrease users' satisfaction, trust, and willingness to continue using a conversational system [15, 58, 101].

Beneteau et al.'s deployment study [14] of Alexa showed that a major barrier for the users to repair conversational breakdowns is that their understandings of the causes of the breakdowns are frequently inaccurate, as a result, the repair strategies they naturally use are often ineffective. Other studies [8, 21, 34, 59, 117, 124]

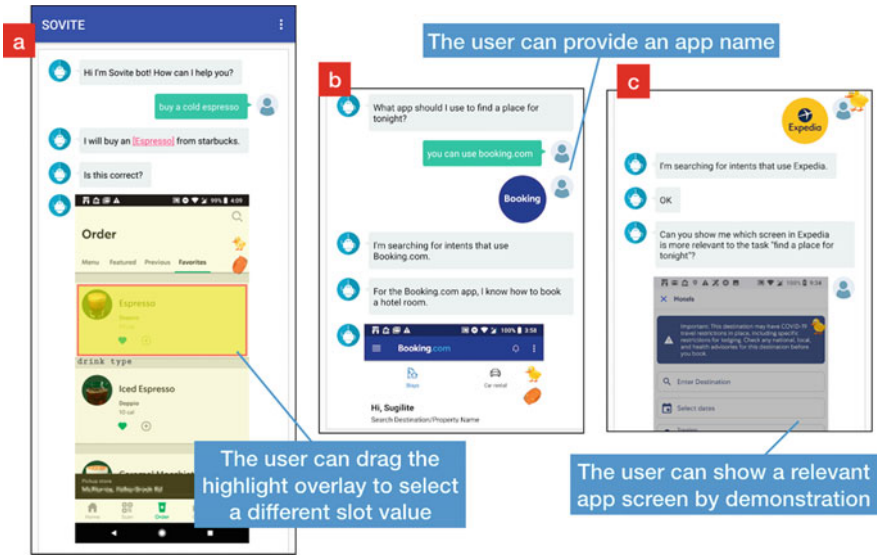

Fig. 5 The interface of SOVITE: **a** SOVITE shows a app GUI screenshot to communicates its state of understanding. The yellow highlight overlay specifies the task slot value. The user can drag the overlay to fix slot value errors. **b** To fix intent detection errors, the user can refer to an app that represents their desired task. SOVITE will match the utterance to an app on the phone (with its icon shown), and look for intents that use or are relevant to this app. **c** If the intent is still ambiguous after referring to an app, the user can show a specific app screen relevant to the desired task

reported similar findings of the types of breakdowns encountered by users and the common repair strategies. In a taxonomy of conversational breakdown repair strategies by Ashktorab et al. [8], repair strategies can be categorized into dimensions of: (1) whether there is evidence of breakdown (i.e., whether the system makes users aware of the breakdown); (2) whether the system attempts to repair (e.g., provide options of potential intents), and (3) whether assistance is provided for user self-repair (e.g., highlight the keywords that contribute to the intent classifier's decision). Among them, the most preferred option by the users was to have the system attempt to help with the repair process by providing options of potential intents. However, as discussed, this approach requires domain-specific "deep knowledge" about the task and error handling flows manually programmed by the developers [5, 107], and therefore, is not practical for user-instructed tasks. The second most preferred strategy in [8] was for the system to provide more transparency into the cause of the breakdown, such as highlighting the keywords that contribute to the results.

Informed by these results, we developed SOVITE,[4] a new interface for SUGILITE that helps users discover, identify the causes of, and recover from conversational breakdowns using a app-grounded multi-modal approach (Fig. 5). Compared with

[4] SOVITE is named after a type of rock. It is also an acronym for **S**ystem for **O**ptimizing **V**oice **I**nterfaces to **T**ackle **E**rrors.

the domain-specific approaches that require "deep knowledge", our approach does not require any additional efforts from the developers. It only requires "shallow knowledge" in a domain-general generic language model to map user intents to the corresponding app screens.

5.5.1 The Design of the Breakdown Handling Interface

Communicating System State with App GUI Screenshots

The first step for SOVITE in supporting the users in repairing conversational breakdowns is to provide transparency into the state of understanding in the system, allowing the users to discover breakdowns and identify their causes. SOVITE leverages the GUI screenshots of mobile apps for this purpose. As shown in Fig. 5a, for the user command, SOVITE displays one or more (when there are multiple slots spanning many screens) screenshots from an app that corresponds to the detected user intent. For intents with slots, it shows screens that contain the GUI widgets corresponding to where the slots would be filled if the task was performed manually using the app GUI. SOVITE also adds a highlight overlay, shown in yellow in Fig. 5a, on top of the app's GUI, which indicates the current slot value. If the slot represents selecting an item from a menu in the GUI, then the corresponding menu item will be highlighted on the screenshot. For an intent without a slot, SOVITE displays the last GUI screen from the procedure of performing the task manually, which usually shows the result of the task. After displaying the screenshot(s), SOVITE asks the user to confirm the understanding of the user's intent. by asking, "I will… `[the task]`, is this correct?", to which the user can verbally respond.

Design Rationale SOVITE's references to app GUIs help with *grounding* in human-agent interactions. In communication theory, the concept of grounding describes conversation as a form of collaborative action to come up with common ground or mutual knowledge [35]. For conversations with computing systems, when the user provides an utterance, the system should provide evidence of understanding so that the user can evaluate the progress toward their goal [26]. As described in the *gulf of evaluation* and *gulf of execution* framework [54, 118] and shown in prior studies of conversational agents [8, 14], execution and evaluation are interdependent—in order to choose an effective strategy for repairing a conversational breakdown, the user needs to first know the current state of understanding in the system and be able to understand the cause of the breakdown. We believe this approach should help users to more effectively identify the understanding errors because it provides better *closeness of mapping* [46] to how the user would naturally approach this task.

Intent Detection Repair with App GUI References

When an intent detection result is incorrect, as evidenced by the wrong app or the wrong functionality of app shown in a confirmation screenshot, or when the agent fails to detect an intent from the user's initial utterance at all (i.e., the system responds

"I don't understand the command."), the user can fix the error by indicating the correct apps and app screens for their desired task.

References to Apps After the user says that the detected intent is incorrect after seeing the app GUI screenshots, or when the system fails to detect an intent, SOVITE asks the user "What app should I use to perform…`[the task]`?", for which the user can say the name of an app for the intended task (shown in Fig. 5b). SOVITE looks up the collection of all supported task intents for not only the intents that *use* this underlying app, but also intents that are semantically *related* to the supplied app.

References to App Screens In certain situations, the user's intent can still be ambiguous after the user indicates the name of an app; there can be multiple intents associated with the app (for example, if the user specifies "Expedia" which can be used for booking flights, cruises, or rental cars), or there can be no supported task intent in the user-provided app and no intent that meets the threshold of being sufficiently "related" to the user-provided app. In these situations, SOVITE will ask the user a follow-up question "Can you show me which screen in…`[the app]` is most relevant to…`[the task]`?" (shown in Fig. 5c). SOVITE then launches the app and asks the user to navigate to the target screen in the app. SOVITE then finds intents that are the most semantically related to this app screen among the ambiguous ones, or asks the user to teach it a new one by demonstration.

Ease of Transition to Out-of-Domain Task Instructions An important advantage of SOVITE's intent disambiguation approach is that it supports the easy transition to the user *instruction* of a new task when the user's intended task is out of scope. An effective approach to support handling out of scope errors is programming-by-demonstration (PBD) [85]. SOVITE's approach can directly connect to the user instruction mode in SUGILITE. Since at this point, SOVITE already knows the most relevant app and app screen for the user's intended task and how to navigate to this screen in the app, it can simply ask the user "Can you teach me how to…`[the task]` using…`[the app]` in this screen", switch back to this screen, and have the user to continue demonstrating the intended task to teach the agent how to fulfill the previously out of scope task intent. The user may also start over and demonstrate from scratch if they do not want to start the instruction from this screen.

Design Rationale The main design rationale of supporting intent detection repairs with app GUI references is to make SOVITE's mechanism of fixing intent detection errors *consistent* with how users discover the errors from SOVITE's display of intent detection results. When users discover the intent detection errors by seeing the wrong apps or the wrong screens displayed in the confirmation screenshots, the most intuitive way for them to fix these errors is to indicate the correct apps and screens that should be used for the intended tasks. Their references to the apps and the screens also allow SOVITE to extract richer semantic context (e.g., the app store descriptions and the text labels found on app GUI screens) than having the user simply rephrase their utterances, helping with finding semantically related task intents.

Fig. 6 SOVITE provides multiple ways to fix text-input slot value errors: *LEFT*: the user can click the corresponding highlight overlay and change its value by adjusting the selection in the original utterance, speaking a new value, or just typing in a new value. *RIGHT*: the user can drag the overlays on the screenshot to move a value to a new slot, or swap the values between two slots

Slot Value Extraction Repair with Direct Manipulation

If the user finds that the intent is correct (i.e., the displayed app and app screen correctly match the user's intended task), but there are errors in the extracted task slot values (i.e., the highlighted textboxes, the values in the highlighted textboxes, or the highlighted menu items on the confirmation screenshots are wrong), the user can fix these errors using direct manipulation on the screenshots.

All the highlight overlays for task slots can be dragged-and-dropped. For slots represented by GUI menu selections, the user can simply drag the highlight overlay to select a different item, as shown in Fig. 5a. The same interaction technique also works for fixing mismatches in the text-input type slot values. For example, if the agent swaps the order between starting location and destination in a "requesting Uber ride" intent, the user can drag these overlays with location names to move them to the right fields in the app GUI screenshot (Fig. 6). When a field is dragged to another field that already has a value, SOVITE performs a *swap* rather than a *replace* so as not to lose any user-supplied data.

Alternatively, when the value for a text-input type slot is incorrect, the user can repair it using the popup dialog shown in Fig. 6. After the user clicks on the highlight overlay for a text-input slot, a dialog will pop up, showing the slot's current value in the user's original utterance. The user can adjust the text selection by dragging the highlight boundaries in the identified entities. The same dialog alternatively allows the user to just enter a new slot value by speech or typing.

Design Rationale We believe these direct manipulation interactions in SOVITE are intuitive to the users. The positions and the contents of the highlight overlays represent where and what slot values would be entered if the task was performed using the GUI of the corresponding app. Therefore, if what SOVITE identified does not match

what the users would do for the intended task, the users can directly fix these *inconsistencies* through simple physical actions such as drag-and-drop and text selection gestures, and see immediate feedback on the screenshots, which are major advantages of direct manipulation [134].

5.6 The Semantic Representation of GUIs

With the rise of data-driven computational methods for modeling user interactions with graphical user interfaces (GUIs), the GUI screens have become not only interfaces for human users to interact with the underlying computing services, but also valuable data sources that encode the underlying task flow, the supported user interactions, and the design patterns of the corresponding apps, which have proven useful for AI-powered applications. For example, programming-by-demonstration (PBD) intelligent agents such as [80, 88, 132] use task-relevant entities and hierarchical structures extracted from GUIs to parameterize, disambiguate, and handle errors in user-demonstrated task automation scripts. ERICA [39] mines a large repository of mobile app GUIs to enable user interface (UI) designers to search for example design patterns to inform their own design. Kite [89] extracts task flows from mobile app GUIs to bootstrap conversational agents.

We present a new self-supervised technique `Screen2Vec` for generating semantic representations of GUI screens and components using their textual content, visual design and layout patterns, and app context metadata. `Screen2Vec`'s approach is inspired by the popular word embedding method `Word2Vec` [111], where the embedding vector representations of GUI screens and components are generated through the process of training a prediction model. But unlike `Word2Vec`, `Screen2Vec` uses a two-layer pipeline informed by the structures of GUIs and GUI interaction traces and incorporates screen- and app-specific metadata.

The embedding vector representations produced by `Screen2Vec` can be used in a variety of useful downstream tasks such as nearest neighbor retrieval, composability-based retrieval, and representing mobile tasks. The self-supervised nature of `Screen2Vec` allows its model to be trained without any manual data labeling efforts—it can be trained with a large collection of GUI screens and the user interaction traces on these screens such as the RICO [38] dataset.

`Screen2Vec` addresses an important gap in prior work about computational HCI research. The lack of comprehensive semantic representations of GUI screens and components has been identified as a major limitation in prior work in GUI-based interactive task learning (e.g., [88, 132]), intelligent suggestive interfaces (e.g., [30]), assistive tools (e.g., [19]), and GUI design aids (e.g., [72, 139]). `Screen2Vec` embeddings can encode the semantics, contexts, layouts, and patterns of GUIs, providing representations of these types of information in a form that can be easily and effectively incorporated into popular modern machine learning models.

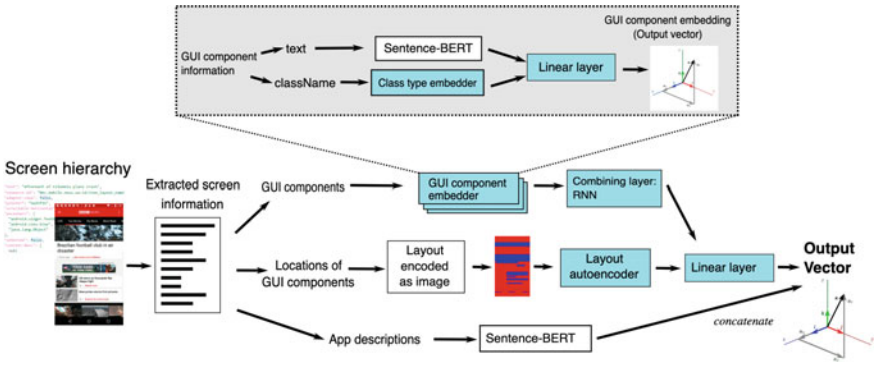

Fig. 7 The two-level architecture of `Screen2Vec` for generating GUI component and screen embeddings. The weights for the steps in teal color are optimized during the training process

5.6.1 `Screen2Vec`'s Approach

Figure 7 illustrates the architecture of `Screen2Vec`. Overall, the pipeline of `Screen2Vec` consists of two levels: the GUI component level (shown in the gray shade) and the GUI screen level. We will describe the approach at a high-level here, you may refer to [87] for the implementation details.

The GUI component level model encodes the textual content and the class type of a GUI component into a 768-dimensional embedding vector to represent the GUI component (e.g., a button, a textbox, a list entry etc.) This GUI component embedding vector is computed with two inputs: (1) a 768-dimensional embedding vector of the text label of the GUI component, encoded using a pre-trained Sentence-BERT [128] model; and (2) a 6-dimensional class embedding vector that represents the class type of the GUI component. The two embedding vectors are combined using a linear layer, resulting in the 768-dimensional GUI component embedding vector that represents the GUI component. The class embeddings in the class type embedder and the weights in the linear layer are optimized through training a Continuous Bag-of-Words (CBOW) prediction task: for each GUI component on each screen, the task predicts the current GUI component using its context (i.e., all the other GUI components on the same screen). The training process optimizes the weights in the class embeddings and the weights in the linear layer for combining the text embedding and the class embedding.

The GUI screen level model encodes the textual content, visual design and layout patterns, and app context of a GUI screen into an 1536-dimensional embedding vector. This GUI screen embedding vector is computed using three inputs: (1) the collection of the GUI component embedding vectors for all the GUI components on the screen (as described in the last paragraph), combined into a 768-dimension vector using a recurrent neural network model (RNN); (2) a 64-dimensional layout embedding vector that encodes the screen's visual layout; and (3) a 768-dimensional embedding vector of the textual App Store description for the underlying app, encoded with

a pre-trained Sentence-BERT [128] model. These GUI and layout vectors are combined using a linear layer, resulting in a 768-dimensional vector. After training, the description embedding vector is concatenated on, resulting in the 1536-dimensional GUI screen embedding vector (if included in the training, the description dominates the entire embedding, overshadowing information specific to that screen within the app). The weights in the RNN layer for combining GUI component embeddings and the weights in the linear layer for producing the final output vector are similarly trained on a CBOW prediction task on a large number of interaction traces (each represented as a sequence of screens). For each trace, a sliding window moves over the sequence of screens. The model tries to use the representation of the context (the surrounding screens) to predict the screen in the middle.

In the training process, we trained `Screen2Vec`[5] on the open-sourced RICO[6] dataset [38]. The RICO dataset contains interaction traces on 66,261 unique GUI screens from 9,384 free Android apps collected using a hybrid crowdsourcing plus automated discovery approach. The models are trained on a cross entropy loss function with an Adam optimizer [63]. In training the GUI screen embedding model, we use *negative sampling* [110, 111] so that we do not have to recalculate and update every screen's embedding on every training iteration, which is computationally expensive and prone to over-fitting. In each iteration, the prediction is compared to the correct screen and a sample of negative data that consists of: a random sampling of size 128 of other screens, the other screens in the batch, and the screens in the same trace as the correct screen, used in the prediction task. We specifically include the screens in the same trace to promote screen-specific learning in this process: This way, we can disincentive screen embeddings that are based solely on the app[7], and emphasize on having the model learn to differentiate the different screens within the same app. You can refer to [87] for details on the training process.

Prediction Task Results

In the screen prediction task, the `Screen2Vec` model performs better than three baseline models (`TextOnly`, `LayoutOnly`, and `VisualOnly`; see [87] for details on the baseline models) in top-1 prediction accuracy, top-k prediction accuracy, and the *normalized* rooted mean square error (RMSE) of the predicted screen embedding vector. See [87] for details on the results and the relevant discussions.

[5] Available at: https://github.com/tobyli/screen2vec.

[6] Available at: http://interactionmining.org/rico.

[7] Since the next screen is always within the same app, and therefore, shares an app description embedding, the prediction task favors having information about the specific app (i.e., app store description embedding) dominate the embedding

5.6.2 Sample Downstream Tasks

Nearest Neighbors

The nearest neighbor task is useful for data-driven design, where the designers want to find examples for inspiration and for understanding the possible design solutions [38]. The task focuses on the similarity between GUI screen embeddings: for a given screen, what are the top-N most similar screens in the dataset? The similar technique can also be used for unsupervised clustering in the dataset to infer different types of GUI screens. In our context, this task also helps demonstrate the different characteristics between `Screen2Vec` and the three baseline models.

We conducted a study with 79 Mechanical Turk workers, where we compared the human-rated similarity of the nearest neighbors results generated by `Screen2Vec` with the baseline models on 5,608 pairs of screen instances. The Mechanical Turk workers rated the nearest neighbor screens generated by the `Screen2Vec` model to be, on average, more similar ($p < 0.0001$) to their source screens than the nearest neighbor screens generated by the baseline models (details on study design and results in [87]).

Subjectively, when looking at the nearest neighbor results, we can see the different aspects of the GUI screens that each different model captures. `Screen2Vec` can create more comprehensive representations that encode the textual content, visual design and layout patterns, and app contexts of the screen compared with the two baselines, which only capture one or two aspects. For example, Fig. 8 shows the example nearest neighbor results for the "request ride" screen in the Lyft app. `Screen2Vec` model retrives the "get direction" screen in the Uber Driver app, "select navigation type" screen in the Waze app, and "request ride" screen in the Free Now (My Taxi) app. Visual and component layout wise, the result screens all feature a menu/information card at the bottom 1/3 to 1/4 of the screen, with a `MapView` taking the majority of the screen space. Content and app domain wise, all these screens are from transportation-related apps that allow the user to configure a trip. In comparison, the `TextOnly` model retrieves the "request ride" screen from the zTrip app, the "main menu" screen from the Hailo app (both zTrip and Hailo are taxi hailing apps), and the home screen of the Paytm app (a mobile payment app in India). The commonality of these screens is that they all include text strings that are semantically similar to "payment" (e.g., add payment type, wallet, pay, add money), and texts that are semantically similar to "destination" and "trips" (e.g., drop off location, trips, bus, flights). But the model neither considers the visual layout and design patterns of the screens, nor the app context. Therefore, the result contains the "main menu" (a quite different type of screen) in the Hailo app and the "home screen" in the Paytm app (a quite different type of screen in a different type of app). The `LayoutOnly` model, on the other hand, retrieves the "exercise logging" screens from the Map My Walk app and the Map My Ride app, and the tutorial screen from the Clever Dialer app. We can see that the content and app-context similarity of the result of the `LayoutOnly` model is quite lower than those of the `Screen2Vec` and `TextOnly` models. However, the result screens all share similar layout features as the source screen, such as the

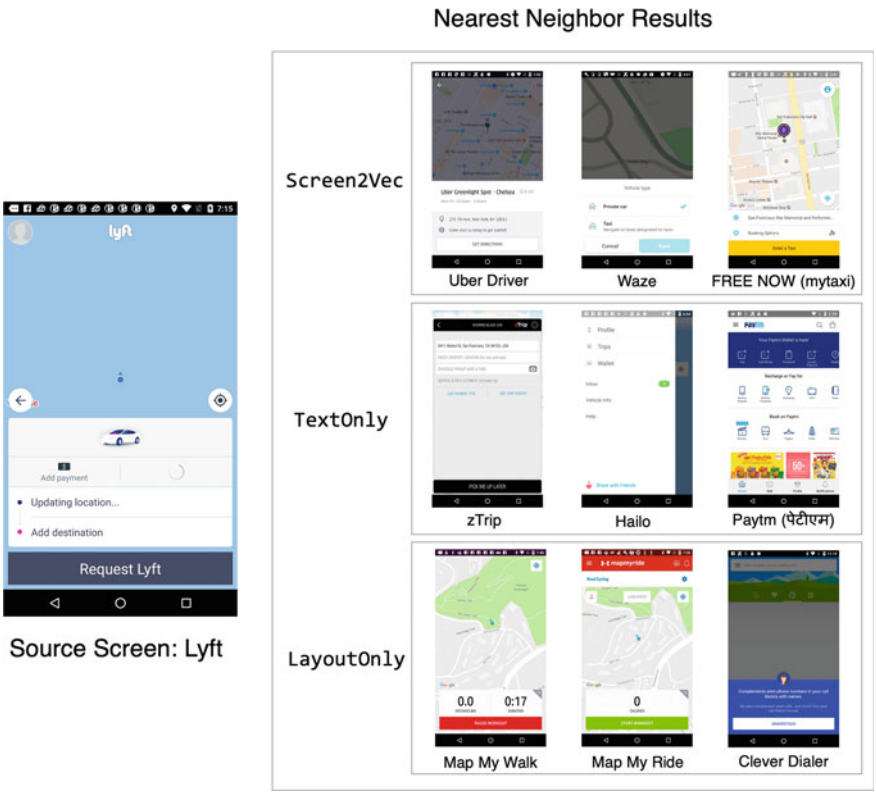

Fig. 8 The example nearest neighbor results for the Lyft "request ride" screen generated by the `Screen2Vec`, `TextOnly`, and `LayoutOnly` models

menu/information card at the bottom of the screen and the screen-wide button at the bottom of the menu (Fig. 8).

Embedding Composability

A useful property of embeddings is that they are composable—meaning that we can add, subtract, and average embeddings to form a meaningful new one. This property is commonly used in word embeddings. For example, in `Word2Vec`, analogies such as "man is to woman as brother is to sister" is reflected in that the vector $(man - woman)$ is similar to the vector $(brother - sister)$. Besides representing analogies, this embedding composability can also be utilized for generative purposes—for example, $(brother - man + woman)$ results in an embedding vector that represents "sister".

This property is also useful in screen embeddings. For example, we can run a nearest neighbor query on the composite embedding of (Marriott app 's "hotel booking" screen + (Cheapoair app's "search result" screen − Cheapoair app's "hotel booking" screen)). The top result is the "search result" screen in the Marriott app.

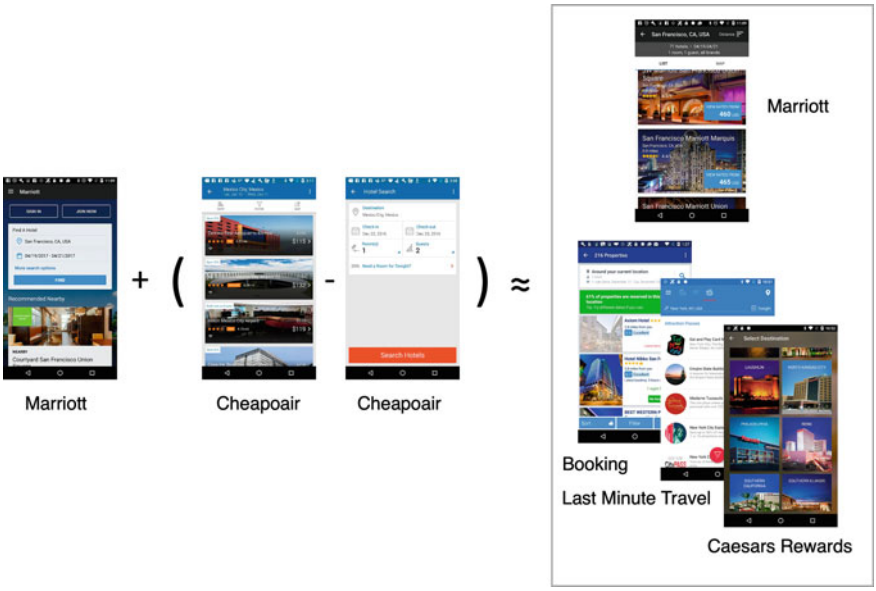

Fig. 9 An example showing the composability of `Screen2Vec` embeddings: running the nearest neighbor query on the composite embedding of (Marriott app 's hotel booking page + Cheapoair app's hotel booking page − Cheapoair app's search result page) can match the Marriott app's search result page, and the similar pages of a few other travel apps

When we filter the result to focus on screens from apps other than Marriott, we get screens that show list results of items from other travel-related apps such as Booking, Last Minute Travel, and Caesars Rewards.

The composability can make `Screen2Vec` particularly useful for GUI design purposes—the designer can leverage the composability to find inspiring examples of GUI designs and layouts.

Screen Embedding Sequences for Representing Mobile Tasks

GUI screens are not only useful data sources individually on their own, but also as building blocks to represent a user's task. A task in an app, or across multiple apps, can be represented as a sequence of GUI screens that makes up the user interaction trace of performing this task through app GUIs. We conduct a preliminary evaluation on the effectiveness of embedding mobile tasks as sequences of `Screen2Vec` screen embedding vectors (details in [87]).

While the task embedding method we explored is quite primitive, it illustrates that the `Screen2Vec` technique can be used to effectively encode mobile tasks into the vector space where semantically similar tasks are close to each other. For the next steps, we plan to further explore this direction. For example, the current method of averaging all the screen embedding vectors does not consider the order

of the screens in the sequence. In the future, we may collect a dataset of human annotations of task similarity, and use techniques that can encode the sequences of items, such as recurrent neural networks (RNN) and long short-term memory (LSTM) networks, to create the task embeddings from sequences of screen embeddings. We may also incorporate the `Screen2Vec` embeddings of the GUI components that were interacted with (e.g., the button that was clicked on) to initiate the screen change into the pipeline for embedding tasks.

5.6.3 Potential Applications of `Screen2Vec`

This section describes several potential applications where the new `Screen2Vec` technique can be useful based on the downstream tasks described in Sect. 5.6.2.

`Screen2Vec` can enable new GUI design aids that take advantage of the nearest neighbor similarity and composability of `Screen2Vec` embeddings. Prior work such as [38, 52, 66] has shown that data-driven tools that enable designers to curate design examples are quite useful for interface designers. Unlike [38], which uses a content-agnostic approach that focuses on the visual and layout similarities, `Screen2Vec` considers the textual content and app metadata in addition to the visual and layout patterns, often leading to different nearest neighbor results as discussed in section. This new type of similarity results will also be useful when focusing on interface design beyond just visual and layout issues, as the results enable designers to query, for example, designs that display similar content or screens that are used in apps in a similar domain. The composability in Screen2Vec embeddings enables querying for design examples at a finer granularity. For example, suppose a designer wishes to find examples for inspiring the design of a new checkout page for app A. They may query for the nearest neighbors of the synthesized embedding App A's order page + (App B's checkout page − App B's order page). Compared with simply querying for the nearest neighbors of App B's checkout page, this synthesized query can encode the interaction context (i.e., the desired page should be the checkout page for App A's order page) in addition to the "checkout" semantics.

The `Screen2Vec` embeddings can also be useful in generative GUI models. Recent models such as the neural design network (NDN) [73] and LayoutGAN [79] can generate realistic GUI layouts based on user-specified constraints (e.g., alignments, relative positions between GUI components). `Screen2Vec` can be used in these generative approaches to incorporate the semantics of GUIs and the contexts of how each GUI screen and component gets used in user interactions. For example, the GUI component prediction model can estimate the likelihood of each GUI component given the context of the other components in a generated screen, providing a heuristic of how likely the GUI components can fit well with each other. Similarly, the GUI screen prediction model may be used as a heuristic to synthesize GUI screens that can better fit with the other screens in the planned user interaction flows. Since `Screen2Vec` has been shown effective in representing mobile tasks in Sect. 5.6.2, where similar tasks will yield similar embeddings, one may also use the task embeddings of performing the same task on an existing app to inform the

generation of new screen designs. The embedding vector form of Screen2Vec representations made them particularly suitable for use in the recent neural network based generative models.

Screen2Vec's capability of embedding tasks can also enhance interactive task learning systems. Specifically, Screen2Vec may be used to enable more powerful procedure generalizations of the learned tasks. We have shown that the Screen2Vec model can effectively predict screens in an interaction trace. Results in Sect. 5.6.2 also indicated that Screen2Vec can embed mobile tasks so that the interaction traces of completing the same task in different apps will be similar to each other in the embedding vector space. Therefore, it is quite promising that Screen2Vec may be used to generalize a task learned from the user by demonstration in one app to another app in the same domain (e.g., generalizing the procedure of ordering coffee in the Starbucks app to the Dunkin' Donut app). In the future, we plan to further explore this direction by incorporating Screen2Vec into open-sourced mobile interactive task learning agents such as SUGILITE.

6 User Evaluations

We conducted several lab user studies to evaluate the usability, efficiency, and effectiveness of SUGILITE. The results of these study showed that end users without significant programming expertise were able to successfully teach the agent the procedures of performing common tasks (e.g., ordering pizza, requesting Uber, checking sports score, ordering coffee) [80], conditional rules for triggering the tasks [88], and concepts relevant to the tasks (e.g., the weather is *hot*, the traffic is *heavy*) [88] using SUGILITE. The users were also able to clarify their intents when ambiguities arise [84] and successfully discover, identify the sources of, and repair conversational breakdowns caused by natural language understanding errors [82]. Most of our participants found SUGILITE easy and natural to use [80, 84, 88]. Efficiency wise, teaching a task usually took the user 3–6 times longer than how long it took to perform the task manually in our studies [80], which indicates that teaching a task using SUGILITE can save time for many repetitive tasks.

7 Limitations

7.1 Platform

SUGILITE and its follow-up work have been developed and tested only on Android phones. SUGILITE retrieves the hierarchical tree structure of the current GUI screen and manipulates the app GUI through Android's Accessibility API. However, the approach used in SUGILITE should apply to any GUI-based apps with hierarchical-

based structures (e.g., the hierarchical DOM structures in web apps). In certain platforms like iOS, while the app GUIs still use hierarchical tree structures, the access to extracting information from and sending inputs to third-party apps has been restricted by the operating system due to security and privacy concerns. In such platforms, implementing a SUGILITE-like system likely requires collaboration with the OS provider (e.g., Apple) or limiting the domain to first-party apps. We also expect working with desktop apps to be more challenging than with mobile apps due to the increased difficulty in inferring their GUI semantics, as the desktop apps often have more complex layouts and more heterogeneous design patterns.

7.2 Runtime Efficiency

An important characteristic of SUGILITE is that it interacts with the underlying third-party app in the same way as a human user do, meaning that it reads information by navigating to the corresponding app screen through the app GUI menu and performs a task by manipulating the app GUI controls. While this approach provides excellent applicability for SUGILITE, allowing the invocation of millions of existing third-party apps without any modification to these apps, it also means that performing a task in SUGILITE is much slower than in an agent that directly invokes the under-the-hood API. It usually takes SUGILITE a few seconds to execute a task automation script. This includes the time needed for SUGILITE to process each screen, plus the extra time for the underlying app to load and to render its GUI.

Another implication of how SUGILITE interacts with the underlying apps is that it needs to run in the foreground of the phone. If an automation script is triggered when the user is actively using the phone at the same time, the user's current task will be interrupted. Similarly, if an external event (e.g., an incoming phone call) interrupts in the middle of executing an automation script, the script execution may fail. One possible way to address the problem is to execute SUGILITE scripts in a virtual machine running in the background, similar to X-Droid [62]. We will leave this for future work.

7.3 Expressiveness

SUGILITE has made several contributions in improving the user expressiveness in programming by demonstration and interactive task learning systems. However, there are still several limitations in SUGILITE's expressiveness, which we plan to address in future work.

The first type of limitations originates from SUGILITE's domain-specific language (DSL) used to specify its automation scripts. For example, it has no support for nested arithmetic operations in the DSL (e.g., one can say "if the price of a Uber ride is greater than 10 dollars" and "if the price of a Uber ride is greater than the price of a

Lyft ride", but not "if the price of a Uber ride is at least 10 dollars more expensive than the price of a Lyft ride.") mostly due to the extra complication in semantic parsing. Correctly parsing the user's natural language description of arithmetic operations into our DSL would likely require a more complicated parsing architecture with a much larger training corpus. It also does not support loops in automation (e.g., "order one of each item in the "Espresso Drinks" category in the Starbucks app"). This is due to SUGILITE's limited capability to capture the internal "states" within the apps and to return to a specific previous state. For example, in the "ordering one of each item" task, the agent needs to return to the GUI state showing the list of items after completing the ordering of the first item in order to order the second item. This cannot be easily done with the current SUGILITE agent. Even if SUGILITE was able to find the "same" (visually similar or have the same activity name) screen, SUGILITE cannot know if the internal state of the underlying app has changed (e.g., adding the first item to the cart affects what other items are available for purchase).

Another limitation in expressiveness is due to the input modalities that SUGILITE tracks in the user demonstrations—it only records a set of common input types (clicks, long-clicks, text entries, etc.) on app GUIs. Gestures (e.g., swipes, flicks), sensory inputs (e.g., tilting or shaking the phone detected by the accelerometer and the gyroscope, auditory inputs from the microphone), and visual inputs (from the phone camera) are not recorded.

7.4 Brittleness

While many measures have been taken to help SUGILITE handle minor changes in app GUIs, SUGILITE scripts can still be brittle after the a change in the underlying app GUI due to either an app update or an external event. As discussed in Sect. 5.2, SUGILITE uses a graph query to locate the correct GUI element to operate on when executing an automation script. Instead of using the absolute (x, y) coordinates for identifying a GUI element like some prior systems do, SUGILITE picks one or more features such as the text label, the ordinal position in a list (e.g., first item in the search result), or the relative position to another GUI element (e.g., the "book" button next to the cheapest flight) that corresponds to the user's intent. Therefore, if a GUI change does not affect the result of the graph query, the automation should still work. In the future, it is possible to further enhance SUGILITE's capability of understanding screen semantics, so that it can automatically detect and handle some of these unexpected screens that do not affect the task without user intervention.

8 Future Work

8.1 Generalization in Programming by Demonstration

Generalization is a central challenge in programming by demonstration [37, 94]. SUGILITE has made several important improvements to the generalization capabilities of the current state-of-art programming by demonstration systems through (1) its multi-modal approach for parameterizing task procedures by combining entities from the user's spoken instructions with information extracted from the hierarchical app GUI structures; and (2) its app-based abstraction model for generalizing learned concepts such as "hot" and "busy" across different apps.

However, there are still opportunities for supporting more powerful generalization. The embedding technique described in Sect. 5.6 opens up the opportunity of cross-app generalization, i.e., when the user has taught performing a task in an app, can the agent generalize the learned procedure to perform a similar task in a different app? Sect. 5.6.2 shows that a task procedure can be represented as a sequence of actions that each consists of (1) the embedding of the screen where the action is performed; and (2) the embedding of the GUI component on which the action is performed, while Sect. 5.6.2 illustrates that it is feasible to find the "equivalence" of a screen in a new app (e.g., locating the search screen in the Cheapoair app based on the search screen in the Marriott app) through arithmetic operations on the screen embeddings. In future work, We plan to explore the design of new mechanisms and their corresponding interfaces that leverage these characteristics of screen embeddings to allow the agent to generalize the learned tasks across different apps with the help from the user.

This approach is inspired by our observation on how human users use unfamiliar apps. In most cases, a user would be able to use an unfamiliar app to perform a task if they have used a similar app before because (1) they have the domain-agnostic knowledge of how mobile apps *generally* work; and (2) they have the app-agnostic knowledge about the task domain. In this planned approach, the domain-agnostic knowledge of app design patterns and layouts is encoded in the app screen embedding model, while the task-domain-specific knowledge can be acquired by the agent through the user's instruction of a similar task in a different app.

Another opportunity for facilitating generalization is to enhance the reasoning of user intents by connecting to large pre-trained commonsense models like COMET [24] and Atomic [131]. While the current SUGLITE agent can be taught new concepts (e.g., hot, busy, and late), procedures (e.g., setting alarms and requesting Uber rides), and if-else rules, the agent does not understand the rationale and reasoning process among these entities (e.g., the user requests a Uber ride when it is late *because* the Uber ride is faster and the user does not want to be late for an event). Understanding such rationale would allow the agent to better generalize user instructions to different contexts and to suggest alternative approaches.

8.2 *Field Study of* SUGILITE

Another future direction is to study the user adoption of SUGILITE through a longitudinal field study. While the usability and the effectiveness of SUGILITE have been validated through task-based lab studies, deploying it to actual users can still be useful for *(i)* further validating the feasibility and robustness of the system in various contexts, *(ii)* measuring the usefulness of SUGILITE in real-life scenarios, and *(iii)* studying the characteristics of how users use SUGILITE. The key goal of the deployment is to study SUGILITE within its intended context of use.

9 Conclusion

We described SUGILITE, a task automation agent that can learn new tasks and relevant concepts interactively from users through their GUI-grounded natural language instructions and demonstrations. This system provides capabilities such as intent clarification, task parameterization, concept generalization, breakdown repairs, and embedding the semantics of GUI screens. SUGILITE shows the promise of using app GUIs for grounding natural language instructions, and the effectiveness of resolving unknown concepts, ambiguities, and vagueness in natural language instructions using a mixed-initiative multi-modal approach.

Acknowledgements This research was supported in part by Verizon through the Yahoo! InMind project, a J.P. Morgan Faculty Research Award, NSF grant IIS-1814472, AFOSR grant FA95501710218, and Google Cloud Research Credits. Any opinions, findings or recommendations expressed here are those of the authors and do not necessarily reflect views of the sponsors. We thank Amos Azaria, Yuanchun Li, Fanglin Chen, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Marissa Radensky, Justin Jia, Kirielle Singarajah, Jingya Chen, Brandon Canfield, Haijun Xia, and Lindsay Popowski for their contributions to this project.

References

1. Adar E, Dontcheva M, Laput G (2014) CommandSpace: modeling the relationships between tasks, descriptions and features. In: Proceedings of the 27th annual ACM symposium on user interface software and technology, UIST '14, pp 167–176. ACM, New York, NY, USA. https://doi.org/10.1145/2642918.2647395. http://doi.acm.org/10.1145/2642918.2647395
2. Alharbi K, Yeh T (2015) Collect, decompile, extract, stats, and diff: mining design pattern changes in android apps. In: Proceedings of the 17th international conference on human-computer interaction with mobile devices and services, MobileHCI '15, pp 515–524. ACM, New York, NY, USA. https://doi.org/10.1145/2785830.2785892. http://doi.acm.org/10.1145/2785830.2785892
3. Allen J, Chambers N, Ferguson G, Galescu L, Jung H, Swift M, Taysom W (2007) PLOW: a collaborative task learning agent. In: Proceedings of the 22Nd national conference on artificial intelligence - volume 2, AAAI'07, pp 1514–1519. AAAI Press, Vancouver, British Columbia, Canada

4. Allen JF, Guinn CI, Horvtz E (1999) Mixed-initiative interaction. IEEE Intell Syst Appl 14(5):14–23
5. Amazon: Alexa Design Guide (2020). https://developer.amazon.com/en-US/docs/alexa/alexa-design/get-started.html
6. Antila V, Polet J, Lämsä A, Liikka J (2012) RoutineMaker: towards end-user automation of daily routines using smartphones. In: 2012 IEEE international conference on pervasive computing and communications workshops (PERCOM workshops), pp 399–402. https://doi.org/10.1109/PerComW.2012.6197519
7. Argall BD, Chernova S, Veloso M, Browning B (2009) A survey of robot learning from demonstration. Robot Auton Syst 57(5):469–483. https://doi.org/10.1016/j.robot.2008.10.024
8. Ashktorab Z, Jain M, Liao QV, Weisz JD (2019) Resilient chatbots: repair strategy preferences for conversational breakdowns. In: Proceedings of the 2019 CHI conference on human factors in computing systems, p 254. ACM
9. Auer S, Bizer C, Kobilarov G, Lehmann J, Cyganiak R, Ives Z (2007) Dbpedia: a nucleus for a web of open data. The semantic web, pp 722–735. http://www.springerlink.com/index/rm32474088w54378.pdf
10. Azaria A, Krishnamurthy J, Mitchell TM (2016) Instructable intelligent personal agent. In: Proceedings of the 30th AAAI conference on artificial intelligence (AAAI), vol 4
11. Ballard BW, Biermann AW (1979) Programming in natural language "NLC" as a prototype. In: Proceedings of the 1979 annual conference, ACM '79, pp 228–237. ACM, New York, NY, USA. https://doi.org/10.1145/800177.810072. http://doi.acm.org/10.1145/800177.810072
12. Banovic N, Grossman T, Matejka J, Fitzmaurice G (2012) Waken: reverse engineering usage information and interface structure from software videos. In: Proceedings of the 25th annual ACM symposium on user interface software and technology, UIST '12, pp 83–92. ACM, New York, NY, USA. https://doi.org/10.1145/2380116.2380129. http://doi.acm.org/10.1145/2380116.2380129
13. Barman S, Chasins S, Bodik R, Gulwani S (2016) Ringer: web automation by demonstration. In: Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications, OOPSLA 2016, pp 748–764. ACM, New York, NY, USA. https://doi.org/10.1145/2983990.2984020. http://doi.acm.org/10.1145/2983990.2984020
14. Beneteau E, Richards OK, Zhang M, Kientz JA, Yip J, Hiniker A (2019) Communication breakdowns between families and alexa. In: Proceedings of the 2019 CHI conference on human factors in computing systems, CHI '19, pp 243:1–243:13. ACM, New York, NY, USA. https://doi.org/10.1145/3290605.3300473. http://doi.acm.org/10.1145/3290605.3300473
15. Bentley F, Luvogt C, Silverman M, Wirasinghe R, White B, Lottridge D (2018) Understanding the long-term use of smart speaker assistants. Proc ACM Interact Mob Wearable Ubiquitous Technol 2(3). https://doi.org/10.1145/3264901
16. Berant J, Chou A, Frostig R, Liang P (2013) Semantic parsing on freebase from question-answer pairs. In: Proceedings of the 2013 conference on empirical methods in natural language processing, pp 1533–1544
17. Bergman L, Castelli V, Lau T, Oblinger D (2005) DocWizards: a system for authoring follow-me documentation wizards. In: Proceedings of the 18th annual ACM symposium on user interface software and technology, UIST '05, pp 191–200. ACM, New York, NY, USA. https://doi.org/10.1145/1095034.1095067. http://doi.acm.org/10.1145/1095034.1095067
18. Biermann AW (1983) Natural Language Programming. In: Biermann AW, Guiho G (eds) Computer program synthesis methodologies, NATO advanced study institutes series. Springer, Netherlands, pp 335–368
19. Bigham JP, Lau T, Nichols J (2009) Trailblazer: enabling blind users to blaze trails through the web. In: Proceedings of the 14th international conference on intelligent user interfaces, IUI '09, pp 177–186. ACM, New York, NY, USA. https://doi.org/10.1145/1502650.1502677
20. Billard A, Calinon S, Dillmann R, Schaal S (2008) Robot programming by demonstration. In: Springer handbook of robotics, pp 1371–1394. Springer. http://link.springer.com/10.1007/978-3-540-30301-5_60

21. Bohus D, Rudnicky AI (2005) Sorry, I didn't catch that!-An investigation of non-understanding errors and recovery strategies. In: 6th SIGdial workshop on discourse and dialogue
22. Bollacker K, Evans C, Paritosh P, Sturge T, Taylor J (2008) Freebase: a collaboratively created graph database for structuring human knowledge. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp 1247–1250. ACM. http://dl.acm.org/citation.cfm?id=1376746
23. Bolt RA (1980) "Put-that-there": voice and gesture at the graphics interface. In: Proceedings of the 7th annual conference on computer graphics and interactive techniques, SIGGRAPH '80, pp 262–270. ACM, New York, NY, USA
24. Bosselut A, Rashkin H, Sap M, Malaviya C, Celikyilmaz A, Choi Y (2019) COMET: commonsense transformers for automatic knowledge graph construction. In: Proceedings of the 57th annual meeting of the association for computational linguistics, pp 4762–4779. ACL, Florence, Italy. https://doi.org/10.18653/v1/P19-1470. https://www.aclweb.org/anthology/P19-1470
25. Brennan SE (1991) Conversation with and through computers. User Model User-Adap Int 1(1):67–86. https://doi.org/10.1007/BF00158952
26. Brennan SE (1998) The grounding problem in conversations with and through computers. Social and cognitive approaches to interpersonal communication, pp 201–225
27. Böhmer M, Hecht B, Schöning J, Krüger A, Bauer G (2011) Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In: Proceedings of the 13th international conference on human computer interaction with mobile devices and services, MobileHCI '11, pp 47–56. ACM, New York, NY, USA. https://doi.org/10.1145/2037373.2037383. http://doi.acm.org/10.1145/2037373.2037383
28. Chai JY, Gao Q, She L, Yang S, Saba-Sadiya S, Xu G (2018) Language to action: towards interactive task learning with physical agents. In: IJCAI, pp 2–9
29. Chandramouli V, Chakraborty A, Navda V, Guha S, Padmanabhan V, Ramjee R (2015) Insider: towards breaking down mobile app silos. In: TRIOS workshop held in conjunction with the SIGOPS SOSP 2015
30. Chen F, Xia K, Dhabalia K, Hong JI (2019) Messageontap: a suggestive interface to facilitate messaging-related tasks. In: Proceedings of the 2019 CHI conference on human factors in computing systems, CHI '19. ACM, New York, NY, USA. https://doi.org/10.1145/3290605.3300805
31. Chen J, Chen C, Xing Z, Xu X, Zhu L, Li G, Wang J (2020) Unblind your apps: predicting natural-language labels for mobile gui components by deep learning. In: Proceedings of the 42nd international conference on software engineering, ICSE '20
32. Chen JH, Weld DS (2008) Recovering from errors during programming by demonstration. In: Proceedings of the 13th international conference on intelligent user interfaces, IUI '08, pp 159–168. ACM, New York, NY, USA. https://doi.org/10.1145/1378773.1378794. http://doi.acm.org/10.1145/1378773.1378794
33. Chkroun M, Azaria A (2019) Lia: a virtual assistant that can be taught new commands by speech. Int J Hum–Comput Interact 1–12
34. Cho J, Rader E (2020) The role of conversational grounding in supporting symbiosis between people and digital assistants. Proc ACM Hum-Comput Interact 4(CSCW1)
35. Clark HH, Brennan SE (1991) Grounding in communication. In: Perspectives on socially shared cognition, pp 127–149. APA, Washington, DC, US. https://doi.org/10.1037/10096-006
36. Cowan BR, Pantidi N, Coyle D, Morrissey K, Clarke P, Al-Shehri S, Earley D, Bandeira N (2017) "what can i help you with?": Infrequent users' experiences of intelligent personal assistants. In: Proceedings of the 19th international conference on human-computer interaction with mobile devices and services, MobileHCI '17, pp 43:1–43:12. ACM, New York, NY, USA. https://doi.org/10.1145/3098279.3098539. http://doi.acm.org/10.1145/3098279.3098539
37. Cypher A, Halbert DC (1993) Watch what I do: programming by demonstration. MIT Press

38. Deka B, Huang Z, Franzen C, Hibschman J, Afergan D, Li Y, Nichols J, Kumar R (2017) Rico: a mobile app dataset for building data-driven design applications. In: Proceedings of the 30th annual ACM symposium on user interface software and technology, UIST '17, pp 845–854. ACM, New York, NY, USA. https://doi.org/10.1145/3126594.3126651. http://doi.acm.org/10.1145/3126594.3126651
39. Deka B, Huang Z, Kumar R (2016) ERICA: interaction mining mobile apps. In: Proceedings of the 29th annual symposium on user interface software and technology, UIST '16, pp 767–776. ACM, New York, NY, USA. https://doi.org/10.1145/2984511.2984581. http://doi.acm.org/10.1145/2984511.2984581
40. Dixon M, Fogarty J (2010) Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '10, pp 1525–1534. ACM, New York, NY, USA. https://doi.org/10.1145/1753326.1753554. http://doi.acm.org/10.1145/1753326.1753554
41. Dixon M, Leventhal D, Fogarty J (2011) Content and hierarchy in pixel-based methods for reverse engineering interface structure. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '11, pp 969–978. ACM, New York, NY, USA. https://doi.org/10.1145/1978942.1979086. http://doi.acm.org/10.1145/1978942.1979086
42. Dixon M, Nied A, Fogarty J (2014) Prefab layers and prefab annotations: extensible pixel-based interpretation of graphical interfaces. In: Proceedings of the 27th annual ACM symposium on user interface software and technology, UIST '14, pp 221–230. ACM, New York, NY, USA. https://doi.org/10.1145/2642918.2647412. http://doi.acm.org/10.1145/2642918.2647412
43. Fast E, Chen B, Mendelsohn J, Bassen J, Bernstein MS (2018) Iris: a conversational agent for complex tasks. In: Proceedings of the 2018 CHI conference on human factors in computing systems, CHI '18, pp 473:1–473:12. ACM, New York, NY, USA. https://doi.org/10.1145/3173574.3174047. http://doi.acm.org/10.1145/3173574.3174047
44. Gao X, Gong R, Zhao Y, Wang S, Shu T, Zhu SC (2020) Joint mind modeling for explanation generation in complex human-robot collaborative tasks. In: 2020 29th IEEE international conference on robot and human interactive communication (RO-MAN), pp 1119–1126. IEEE
45. Gluck KA, Laird JE (2019) Interactive task learning: humans, robots, and agents acquiring new tasks through natural interactions, vol 26. MIT Press
46. Green TR (1989) Cognitive dimensions of notations. People and Computers V pp 443–460. https://books.google.com/books?hl=en&lr=&id=BTxOtt4X920C&oi=fnd&pg=PA443&dq=Cognitive+dimensions+of+notations&ots=OEqg1By_Rj&sig=dpg1zZFRHpBVC_r0--XLyLr6718
47. Grudin J, Jacques R (2019) Chatbots, humbots, and the quest for artificial general intelligence. In: Proceedings of the 2019 CHI conference on human factors in computing systems, pp 1–11
48. Guo A, Kong J, Rivera M, Xu FF, Bigham JP (2019) StateLens: a reverse engineering solution for making existing dynamic touchscreens accessible. In: Proceedings of the 32nd annual ACM symposium on user interface software and technology (UIST 2019), p 15
49. Gur I, Yavuz S, Su Y, Yan X (2018) DialSQL: dialogue based structured query generation. In: Proceedings of the 56th annual meeting of the association for computational linguistics (volume 1: long papers), pp 1339–1349. ACL, Melbourne, Australia. https://doi.org/10.18653/v1/P18-1124. https://www.aclweb.org/anthology/P18-1124
50. Hartmann B, Wu L, Collins K, Klemmer SR (2007) Programming by a sample: rapidly creating web applications with d.mix. In: Proceedings of the 20th annual ACM symposium on user interface software and technology, UIST '07, pp 241–250. ACM, New York, NY, USA. https://doi.org/10.1145/1294211.1294254. http://doi.acm.org/10.1145/1294211.1294254
51. Horvitz E (1999) Principles of mixed-initiative user interfaces. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '99, pp 159–166. ACM, New York, NY, USA. https://doi.org/10.1145/302979.303030
52. Huang F, Canny JF, Nichols J (2019) Swire: sketch-based user interface retrieval. In: Proceedings of the 2019 CHI conference on human factors in computing systems, CHI '19, pp 1–10. ACM, New York, NY, USA. https://doi.org/10.1145/3290605.3300334

53. Huang THK, Azaria A, Bigham JP (2016) InstructableCrowd: creating IF-THEN rules via conversations with the crowd, pp 1555–1562. ACM Press. https://doi.org/10.1145/2851581.2892502. http://dl.acm.org/citation.cfm?doid=2851581.2892502
54. Hutchins EL, Hollan JD, Norman DA (1986) Direct manipulation interfaces
55. Iba S, Paredis CJJ, Khosla PK (2005) Interactive multimodal robot programming. Int J Robot Res 24(1):83–104. https://doi.org/10.1177/0278364904049250
56. IFTTT (2016) IFTTT: connects the apps you love. https://ifttt.com/
57. Intharah T, Turmukhambetov D, Brostow GJ (2019) Hilc: domain-independent pbd system via computer vision and follow-up questions. ACM Trans Interact Intell Syst 9(2-3):16:1–16:27. https://doi.org/10.1145/3234508. http://doi.acm.org/10.1145/3234508
58. Jain M, Kumar P, Kota R, Patel SN (2018) Evaluating and informing the design of chatbots. In: Proceedings of the 2018 designing interactive systems conference, pp 895–906. ACM
59. Jiang J, Jeng W, He D (2013) How do users respond to voice input errors?: lexical and phonetic query reformulation in voice search. In: Proceedings of the 36th international ACM SIGIR conference on research and development in information retrieval, pp 143–152. ACM
60. Kasturi T, Jin H, Pappu A, Lee S, Harrison B, Murthy R, Stent A (2015) The cohort and speechify libraries for rapid construction of speech enabled applications for android. In: Proceedings of the 16th annual meeting of the special interest group on discourse and dialogue, pp 441–443
61. Kate RJ, Wong YW, Mooney RJ (2005) Learning to transform natural to formal languages. In: Proceedings of the 20th national conference on artificial intelligence - volume 3, AAAI'05, pp 1062–1068. AAAI Press, Pittsburgh, Pennsylvania. http://dl.acm.org/citation.cfm?id=1619499.1619504
62. Kim D, Park S, Ko J, Ko SY, Lee SJ (2019) X-droid: a quick and easy android prototyping framework with a single-app illusion. In: Proceedings of the 32nd annual ACM symposium on user interface software and technology, UIST '19, pp 95–108. ACM, New York, NY, USA. https://doi.org/10.1145/3332165.3347890
63. Kingma DP, Ba J (2015) Adam: a method for stochastic optimization. In: Bengio Y, LeCun Y (eds) 3rd international conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings. http://arxiv.org/abs/1412.6980
64. Kirk J, Mininger A, Laird J (2016) Learning task goals interactively with visual demonstrations. Biol Inspired Cogn Archit 18:1–8
65. Ko AJ, Abraham R, Beckwith L, Blackwell A, Burnett M, Erwig M, Scaffidi C, Lawrance J, Lieberman H, Myers B, Rosson MB, Rothermel G, Shaw M, Wiedenbeck S (2011) The state of the art in end-user software engineering. ACM Comput Surv 43(3), 21:1–21:44. https://doi.org/10.1145/1922649.1922658. http://doi.acm.org/10.1145/1922649.1922658
66. Kumar R, Satyanarayan A, Torres C, Lim M, Ahmad S, Klemmer SR, Talton JO (2013) Webzeitgeist: design mining the web. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '13, pp 3083–3092. ACM, New York, NY, USA. https://doi.org/10.1145/2470654.2466420
67. Kurihara K, Goto M, Ogata J, Igarashi T (2006) Speech pen: predictive handwriting based on ambient multimodal recognition. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp 851–860. ACM
68. Labutov I, Srivastava S, Mitchell T (2018) Lia: a natural language programmable personal assistant. In: Proceedings of the 2018 conference on empirical methods in natural language processing: system demonstrations, pp 145–150
69. Laird JE, Gluck K, Anderson J, Forbus KD, Jenkins OC, Lebiere C, Salvucci D, Scheutz M, Thomaz A, Trafton G, Wray RE, Mohan S, Kirk JR (2017) Interactive task learning. IEEE Intell Syst 32(4):6–21. https://doi.org/10.1109/MIS.2017.3121552
70. Laput GP, Dontcheva M, Wilensky G, Chang W, Agarwala A, Linder J, Adar E (2013) PixelTone: a multimodal interface for image editing. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '13, pp 2185–2194. ACM, New York, NY, USA. https://doi.org/10.1145/2470654.2481301. http://doi.acm.org/10.1145/2470654.2481301

71. Lau T (2009) Why programming-by-demonstration systems fail: lessons learned for usable AI. AI Mag 30(4):65–67. http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262
72. Lee C, Kim S, Han D, Yang H, Park YW, Kwon BC, Ko S (2020) Guicomp: a gui design assistant with real-time, multi-faceted feedback. In: Proceedings of the 2020 CHI conference on human factors in computing systems, CHI '20, pp 1–13. ACM, New York, NY, USA. https://doi.org/10.1145/3313831.3376327
73. Lee HY, Yang W, Jiang L, Le M, Essa I, Gong H, Yang MH (2020) Neural design network: graphic layout generation with constraints. In: European conference on computer vision (ECCV)
74. Lee TY, Dugan C, Bederson BB (2017) Towards understanding human mistakes of programming by example: an online user study. In: Proceedings of the 22nd international conference on intelligent user interfaces, IUI '17, pp 257–261. ACM, New York, NY, USA. https://doi.org/10.1145/3025171.3025203. http://doi.acm.org/10.1145/3025171.3025203
75. Leshed G, Haber EM, Matthews T, Lau T (2008) CoScripter: automating & sharing how-to knowledge in the enterprise. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '08, pp 1719–1728. ACM, New York, NY, USA. https://doi.org/10.1145/1357054.1357323. http://doi.acm.org/10.1145/1357054.1357323
76. Li F, Jagadish HV (2014) Constructing an interactive natural language interface for relational databases. Proc VLDB Endow 8(1):73–84. https://doi.org/10.14778/2735461.2735468
77. Li H, Wang YP, Yin J, Tan G (2019) Smartshell: automated shell scripts synthesis from natural language. Int J Softw Eng Knowl Eng 29(02):197–220
78. Li I, Nichols J, Lau T, Drews C, Cypher A (2010) Here's What I Did: sharing and reusing web activity with ActionShot. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '10, pp 723–732. ACM, New York, NY, USA. https://doi.org/10.1145/1753326.1753432. http://doi.acm.org/10.1145/1753326.1753432
79. Li J, Yang J, Hertzmann A, Zhang J, Xu T (2019) Layoutgan: synthesizing graphic layouts with vector-wireframe adversarial networks. IEEE Trans Pattern Anal Mach Intell
80. Li TJJ, Azaria A, Myers BA (2017) SUGILITE: creating multimodal smartphone automation by demonstration. In: Proceedings of the 2017 CHI conference on human factors in computing systems, CHI '17, pp 6038–6049. ACM, New York, NY, USA. https://doi.org/10.1145/3025453.3025483. http://doi.acm.org/10.1145/3025453.3025483
81. Li TJJ, Chen J, Canfield B, Myers BA (2020) Privacy-preserving script sharing in gui-based programming-by-demonstration systems. Proc ACM Hum-Comput Interact 4(CSCW1). https://doi.org/10.1145/3392869
82. Li TJJ, Chen J, Xia H, Mitchell TM, Myers BA (2020) Multi-modal repairs of conversational breakdowns in task-oriented dialogs. In: Proceedings of the 33rd annual ACM symposium on user interface software and technology, UIST 2020. ACM. https://doi.org/10.1145/3379337.3415820
83. Li TJJ, Hecht B (2014) WikiBrain: making computer programs smarter with knowledge from wikipedia
84. Li TJJ, Labutov I, Li XN, Zhang X, Shi W, Mitchell TM, Myers BA (2018) APPINITE: a multi-modal interface for specifying data descriptions in programming by demonstration using verbal instructions. In: Proceedings of the 2018 IEEE symposium on visual languages and human-centric computing (VL/HCC 2018)
85. Li TJJ, Labutov I, Myers BA, Azaria A, Rudnicky AI, Mitchell TM (2018) Teaching agents when they fail: end user development in goal-oriented conversational agents. In: Studies in conversational UX design. Springer
86. Li TJJ, Li Y, Chen F, Myers BA (2017) Programming IoT devices by demonstration using mobile apps. In: Barbosa S, Markopoulos P, Paterno F, Stumpf S, Valtolina S (eds) End-user development. Springer, Cham, pp 3–17
87. Li TJJ, Popowski L, Mitchell TM, Myers BA (2021) Screen2vec: semantic embedding of gui screens and gui components. In: Proceedings of the 2021 CHI conference on human factors in computing systems, CHI '21. ACM

88. Li TJJ, Radensky M, Jia J, Singarajah K, Mitchell TM, Myers BA (2019) PUMICE: a multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In: Proceedings of the 32nd annual ACM symposium on user interface software and technology (UIST 2019), UIST 2019. ACM. https://doi.org/10.1145/3332165.3347899
89. Li TJJ, Riva O (2018) KITE: building conversational bots from mobile apps. In: Proceedings of the 16th ACM international conference on mobile systems, applications, and services (MobiSys 2018). ACM
90. Li Y, He J, Zhou X, Zhang Y, Baldridge J (2020) Mapping natural language instructions to mobile UI action sequences. In: Proceedings of the 58th annual meeting of the association for computational linguistics, pp 8198–8210. ACL, Online. https://doi.org/10.18653/v1/2020.acl-main.729. https://www.aclweb.org/anthology/2020.acl-main.729
91. Li Y, Li G, He L, Zheng J, Li H, Guan Z (2020) Widget captioning: generating natural language description for mobile user interface elements. In: Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP), pp 5495–5510. ACL, Online. https://doi.org/10.18653/v1/2020.emnlp-main.443. https://www.aclweb.org/anthology/2020.emnlp-main.443
92. Liang P (2016) Learning executable semantic parsers for natural language understanding. Commun ACM 59(9):68–76
93. Liang P, Jordan MI, Klein D (2013) Learning dependency-based compositional semantics. Comput Linguist 39(2):389–446
94. Lieberman H (2001) Your wish is my command: programming by example. Morgan Kaufmann
95. Lieberman H, Liu H (2006) Feasibility studies for programming in natural language. In: End user development, pp 459–473. Springer
96. Lieberman H, Maulsby D (1996) Instructible agents: software that just keeps getting better. IBM Syst J 35(3.4):539–556. https://doi.org/10.1147/sj.353.0539
97. Lin J, Wong J, Nichols J, Cypher A, Lau TA (2009) End-user programming of mashups with vegemite. In: Proceedings of the 14th international conference on intelligent user interfaces, IUI '09, pp 97–106. ACM, New York, NY, USA. https://doi.org/10.1145/1502650.1502667. http://doi.acm.org/10.1145/1502650.1502667
98. Liu EZ, Guu K, Pasupat P, Shi T, Liang P (2018) Reinforcement learning on web interfaces using workflow-guided exploration. CoRR. http://arxiv.org/abs/1802.08802
99. Liu TF, Craft M, Situ J, Yumer E, Mech R, Kumar R (2018) Learning design semantics for mobile apps. In: Proceedings of the 31st annual ACM symposium on user interface software and technology, UIST '18, pp 569–579. ACM, New York, NY, USA. https://doi.org/10.1145/3242587.3242650
100. LlamaLab: Automate: everyday automation for Android (2016). http://llamalab.com/automate/
101. Luger E, Sellen A (2016) "like having a really bad pa": the gulf between user expectation and experience of conversational agents. In: Proceedings of the 2016 CHI conference on human factors in computing systems, CHI '16, pp 5286–5297. ACM, New York, NY, USA. https://doi.org/10.1145/2858036.2858288. http://doi.acm.org/10.1145/2858036.2858288
102. Maes P (1994) Agents that reduce work and information overload. Commun ACM 37(7):30–40. https://doi.org/10.1145/176789.176792. http://doi.acm.org/10.1145/176789.176792
103. Mankoff J, Abowd GD, Hudson SE (2000) Oops: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. Comput Graph 24(6):819–834
104. Marin R, Sanz PJ, Nebot P, Wirz R (2005) A multimodal interface to control a robot arm via the web: a case study on remote programming. IEEE Trans Ind Electron 52(6):1506–1520. https://doi.org/10.1109/TIE.2005.858733
105. Maués RDA, Barbosa SDJ (2013) Keep doing what i just did: automating smartphones by demonstration. In: Proceedings of the 15th international conference on human-computer interaction with mobile devices and services, MobileHCI '13, pp 295–303. ACM, New York, NY, USA. https://doi.org/10.1145/2493190.2493216. http://doi.acm.org/10.1145/2493190.2493216

106. McDaniel RG, Myers BA (1999) Getting more out of programming-by-demonstration. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '99, pp 442–449. ACM, New York, NY, USA. https://doi.org/10.1145/302979.303127. http://doi.acm.org/10.1145/302979.303127
107. McTear M, O'Neill I, Hanna P, Liu X (2005) Handling errors and determining confirmation strategies–an object-based approach. Speech Commun 45(3):249–269. https://doi.org/10.1016/j.specom.2004.11.006. http://www.sciencedirect.com/science/article/pii/S0167639304001426. Special Issue on Error Handling in Spoken Dialogue Systems
108. Menon A, Tamuz O, Gulwani S, Lampson B, Kalai A (2013) A machine learning framework for programming by example, pp 187–195. http://machinelearning.wustl.edu/mlpapers/papers/ICML2013_menon13
109. Mihalcea R, Liu H, Lieberman H (2006) NLP (Natural Language Processing) for NLP (Natural Language Programming). In: Gelbukh A (ed) Computational linguistics and intelligent text processing. Lecture notes in computer science. Springer, Berlin, Heidelberg, pp 319–330
110. Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. arXiv:1301.3781 [cs]. http://arxiv.org/abs/1301.3781. ArXiv: 1301.3781
111. Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality
112. Mohan S, Laird JE (2014) Learning goal-oriented hierarchical tasks from situated interactive instruction. In: Proceedings of the twenty-eighth AAAI conference on artificial intelligence, AAAI'14, pp 387–394. AAAI Press
113. Myers B, Malkin R, Bett M, Waibel A, Bostwick B, Miller RC, Yang J, Denecke M, Seemann E, Zhu J et al (2002) Flexi-modal and multi-machine user interfaces. In: Proceedings of the fourth IEEE international conference on multimodal interfaces, pp 343–348. IEEE
114. Myers BA (1986) Visual programming, programming by example, and program visualization: a taxonomy. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '86, pp 59–66. ACM, New York, NY, USA. https://doi.org/10.1145/22627.22349. http://doi.acm.org/10.1145/22627.22349
115. Myers BA, Ko AJ, Scaffidi C, Oney S, Yoon Y, Chang K, Kery MB, Li TJJ (2017) Making end user development more natural. In: New perspectives in end-user development, pp 1–22. Springer, Cham. https://doi.org/10.1007/978-3-319-60291-2_1. https://link.springer.com/chapter/10.1007/978-3-319-60291-2_1
116. Myers BA, McDaniel R (2001) Sometimes you need a little intelligence, sometimes you need a lot. Your wish is my command: programming by example. Morgan Kaufmann Publishers, San Francisco, CA, pp 45–60. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.8085&rep=rep1&type=pdf
117. Myers C, Furqan A, Nebolsky J, Caro K, Zhu J (2018) Patterns for how users overcome obstacles in voice user interfaces. In: Proceedings of the 2018 CHI conference on human factors in computing systems, pp 1–7
118. Norman D (2013) The design of everyday things: revised and expanded edition. Basic Books
119. Oviatt S (1999) Mutual disambiguation of recognition errors in a multimodel architecture. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp 576–583. ACM
120. Oviatt S (1999) Ten myths of multimodal interaction. Commun ACM 42(11):74–81 https://doi.org/10.1145/319382.319398. http://doi.acm.org/10.1145/319382.319398
121. Oviatt S, Cohen P (2000) Perceptual user interfaces: multimodal interfaces that process what comes naturally. Commun ACM 43(3):45–53
122. Pasupat P, Jiang TS, Liu E, Guu K, Liang P (2018) Mapping natural language commands to web elements. In: Proceedings of the 2018 conference on empirical methods in natural language processing, pp 4970–4976. ACL, Brussels, Belgium. https://doi.org/10.18653/v1/D18-1540. https://www.aclweb.org/anthology/D18-1540

123. Pasupat P, Liang P (2015) Compositional semantic parsing on semi-structured tables. In: Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing. http://arxiv.org/abs/1508.00305. ArXiv: 1508.00305
124. Porcheron M, Fischer JE, Reeves S, Sharples S (2018) Voice interfaces in everyday life. In: Proceedings of the 2018 CHI conference on human factors in computing systems, CHI '18. ACM, New York, NY, USA. https://doi.org/10.1145/3173574.3174214
125. Price D, Rilofff E, Zachary J, Harvey B (2000) NaturalJava: a natural language interface for programming in java. In: Proceedings of the 5th international conference on intelligent user interfaces, IUI '00, pp 207–211. ACM, New York, NY, USA. https://doi.org/10.1145/325737.325845. http://doi.acm.org/10.1145/325737.325845
126. Qi S, Jia B, Huang S, Wei P, Zhu SC (2020) A generalized earley parser for human activity parsing and prediction. IEEE Trans Pattern Anal Mach Intell
127. Ravindranath L, Thiagarajan A, Balakrishnan H, Madden S (2012) Code in the air: simplifying sensing and coordination tasks on smartphones. In: Proceedings of the twelfth workshop on mobile computing systems & applications, HotMobile '12, pp 4:1–4:6. ACM, New York, NY, USA. https://doi.org/10.1145/2162081.2162087. http://doi.acm.org/10.1145/2162081.2162087
128. Reimers N, Gurevych I (2019) Sentence-bert: sentence embeddings using siamese bert-networks. In: Proceedings of the 2019 conference on empirical methods in natural language processing. ACL. http://arxiv.org/abs/1908.10084
129. Rodrigues A (2015) Breaking barriers with assistive macros. In: Proceedings of the 17th international ACM SIGACCESS conference on computers & accessibility, ASSETS '15, pp 351–352. ACM, New York, NY, USA. https://doi.org/10.1145/2700648.2811322. http://doi.acm.org/10.1145/2700648.2811322
130. Sahami Shirazi A, Henze N, Schmidt A, Goldberg R, Schmidt B, Schmauder H (2013) Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, EICS '13, pp 275–284. ACM, New York, NY, USA. https://doi.org/10.1145/2494603.2480308. http://doi.acm.org/10.1145/2494603.2480308
131. Sap M, Le Bras R, Allaway E, Bhagavatula C, Lourie N, Rashkin H, Roof B, Smith NA, Choi Y (2019) Atomic: an atlas of machine commonsense for if-then reasoning. Proc AAAI Conf Artif Intell 33:3027–3035
132. Sereshkeh AR, Leung G, Perumal K, Phillips C, Zhang M, Fazly A, Mohomed I (2020) Vasta: a vision and language-assisted smartphone task automation system. In: Proceedings of the 25th international conference on intelligent user interfaces, pp 22–32
133. She L, Chai J (2017) Interactive learning of grounded verb semantics towards human-robot communication. In: Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: long papers), pp 1634–1644. ACL, Vancouver, Canada. https://doi.org/10.18653/v1/P17-1150. https://www.aclweb.org/anthology/P17-1150
134. Shneiderman B (1983) Direct manipulation: a step beyond programming languages. Computer 16(8):57–69. https://doi.org/10.1109/MC.1983.1654471
135. Shneiderman B, Plaisant C, Cohen M, Jacobs S, Elmqvist N, Diakopoulos N (2016) Designing the user interface: strategies for effective human-computer interaction, 6, edition. Pearson, Boston
136. Srivastava S, Labutov I, Mitchell T (2017) Joint concept learning and semantic parsing from natural language explanations. In: Proceedings of the 2017 conference on empirical methods in natural language processing, pp 1527–1536
137. Su Y, Hassan Awadallah A, Wang M, White RW (2018) Natural language interfaces with fine-grained user interaction: a case study on web apis. In: The 41st international ACM SIGIR conference on research and development in information retrieval, SIGIR '18, pp 855–864. ACM, New York, NY, USA. https://doi.org/10.1145/3209978.3210013
138. Suhm B, Myers B, Waibel A (2001) Multimodal error correction for speech user interfaces. ACM Trans Comput-Hum Interact 8(1):60–98. https://doi.org/10.1145/371127.371166. http://doi.acm.org/10.1145/371127.371166

139. Swearngin A, Dontcheva M, Li W, Brandt J, Dixon M, Ko AJ (2018) Rewire: interface design assistance from examples. In: Proceedings of the 2018 CHI conference on human factors in computing systems, CHI '18, pp 1–12. ACM, New York, NY, USA. https://doi.org/10.1145/3173574.3174078
140. Ur B, McManus E, Pak Yong Ho M, Littman ML (2014) Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '14, pp 803–812. ACM, New York, NY, USA. https://doi.org/10.1145/2556288.2557420. http://doi.acm.org/10.1145/2556288.2557420
141. Vadas D, Curran JR (2005) Programming with unrestricted natural language. In: Proceedings of the Australasian language technology workshop 2005, pp 191–199
142. Vrandečić D, Krötzsch M (2014) Wikidata: a free collaborative knowledgebase. Commun ACM 57(10):78–85. http://dl.acm.org/citation.cfm?id=2629489
143. Xu Q, Erman J, Gerber A, Mao Z, Pang J, Venkataraman S (2011) Identifying diverse usage behaviors of smartphone apps. In: Proceedings of the 2011 ACM SIGCOMM conference on internet measurement conference, IMC '11, pp 329–344. ACM, New York, NY, USA. https://doi.org/10.1145/2068816.2068847. http://doi.acm.org/10.1145/2068816.2068847
144. Yang JJ, Lam MS, Landay JA (2020) Dothishere: multimodal interaction to improve cross-application tasks on mobile devices. In: Proceedings of the 33rd annual ACM symposium on user interface software and technology, UIST '20, pp 35–44. ACM, New York, NY, USA. https://doi.org/10.1145/3379337.3415841
145. Yao Z, Su Y, Sun H, Yih WT (2019) Model-based interactive semantic parsing: a unified framework and a text-to-SQL case study. In: Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP), pp 5447–5458. ACL, Hong Kong, China. https://doi.org/10.18653/v1/D19-1547. https://www.aclweb.org/anthology/D19-1547
146. Yao Z, Tang Y, Yih WT, Sun H, Su Y (2020) An imitation game for learning semantic parsers from user interaction. In: Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP), pp 6883–6902. ACL, Online. https://doi.org/10.18653/v1/2020.emnlp-main.559. https://www.aclweb.org/anthology/2020.emnlp-main.559
147. Yeh T, Chang TH, Miller RC (2009) Sikuli: using GUI screenshots for search and automation. In: Proceedings of the 22nd annual ACM symposium on user interface software and technology, UIST '09, pp 183–192. ACM, New York, NY, USA. https://doi.org/10.1145/1622176.1622213. http://doi.acm.org/10.1145/1622176.1622213
148. Zhang X, Ross AS, Fogarty J (2018) Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement. In: Proceedings of the 31st annual ACM symposium on user interface software and technology, UIST '18
149. Zhang Z, Zhu Y, Zhu SC (2020) Graph-based hierarchical knowledge representation for robot task transfer from virtual to physical world. In: 2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)
150. Zhao S, Ramos J, Tao J, Jiang Z, Li S, Wu Z, Pan G, Dey AK (2016) Discovering different kinds of smartphone users through their application usage behaviors. In: Proceedings of the 2016 ACM international joint conference on pervasive and ubiquitous computing, UbiComp '16, pp 498–509. ACM, New York, NY, USA. https://doi.org/10.1145/2971648.2971696. http://doi.acm.org/10.1145/2971648.2971696