A Multi-Modal Intelligent Agent that Learns from Demonstrations and Natural Language Instructions

Toby Jia-Jun Li
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

tobyli@cs.cmu.edu
http://toby.li/

CMU-HCII-21-102

May 3, 2021

Thesis Committee:

Brad A. Myers (Chair), Carnegie Mellon University
Tom M. Mitchell, Carnegie Mellon University
Jeffrey P. Bigham, Carnegie Mellon University
John Zimmerman, Carnegie Mellon University
Philip J. Guo, University of California San Diego

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: end user development, end user programming, interactive task learning, programming by demonstration, programming by example, multi-modal interaction, verbal instruction, natural language programming, task automation, intelligent agent, instructable agent, conversational assistant, human-AI interaction, human-AI collaboration.

Abstract

Intelligent agents that can perform tasks on behalf of users have become increasingly popular with the growing ubiquity of "smart" devices such as phones, wearables, and smart home devices. They allow users to automate common tasks and to perform tasks in contexts where the direct manipulation of traditional graphical user interfaces (GUIs) is infeasible or inconvenient. However, the capabilities of such agents are limited by their available skills (i.e., the procedural knowledge of *how* to do something) and conceptual knowledge (i.e., *what* does a concept mean). Most current agents (e.g., Siri, Google Assistant, Alexa) either have fixed sets of capabilities or mechanisms that allow only skilled third-party developers to extend agent capabilities. As a result, they fall short in supporting "long-tail" tasks and suffer from the lack of customizability and flexibility.

To address this problem, I and my collaborators designed SUGILITE, a new intelligent agent that allows end users to teach new tasks and concepts in a natural way. SUGILITE uses a *multi-modal* approach that combines *programming by demonstration* (PBD) and learning from natural language instructions to support end-user development for intelligent agents. The lab usability evaluation results showed that the prototype of SUGILITE allowed users with little or no programming expertise to successfully teach the agent common smartphone tasks such as ordering coffee, booking restaurants, and checking sports scores, as well as the appropriate conditionals for triggering these actions on user-defined situations using user-taught concepts for determining these conditions. My dissertation presents a new human-AI interaction paradigm for interactive task learning, where the existing third-party app GUIs are used as a medium for users to communicate their intents with an AI agent in addition to being the interface for interacting with the underlying computing services.

Through the development of the integrated SUGILITE system over the past five years, this dissertation presents seven main technical contributions including: *(i)* a new approach to allow the agent to generalize from learned task procedures by inferring task parameters and their associated possible values from verbal instructions and mobile app GUIs, *(ii)* a new method to address the *data description problem* in PBD by allowing users to verbally explain ambiguous or vague demonstrated actions, *(iii)* a new multi-modal interface to enable users to teach the conceptual knowledge used in conditionals to the agent, *(iv)* a new mechanism to extend mobile app based PBD to smart home and Internet of Things (IoT) automation, *(v)* a new multi-modal interface that helps users discover, identify the causes of, and recover from conversational breakdowns using existing mobile app GUIs for grounding, *(vi)* a new privacy-preserving approach that can identify and obfuscate the potential personal information in GUI-based PBD scripts based on the uniqueness of information entries with respect to the corresponding app GUI context, and *(vii)* a new self-supervised technique for generating semantic representations of GUI screens and components in embedding vectors without requiring manual annotation.

Acknowledgements

TBD

Contents

List of Figures

List of Tables

List of Acronyms

Following a tradition[1] started by my doctoral advisor Brad Myers, the names of most of the systems and subsystems described in this dissertation are acronyms based on gemstones or rocks. Below is a list of these acronyms and what they stand for, as well as references to the corresponding chapters in this dissertation.

1. SUGILITE
   Smartphone **U**sers **G**enerating **I**ntelligent **L**ikeable **I**nterfaces **T**hrough **E**xamples
   An end-user-programmable intelligent agent that can learn new tasks from the user through a combination of app demonstrations and natural language instructions. (Chapter 3)

2. APPINITE
   Automation **P**rogramming on **P**hone **I**nterfaces using **N**atural-language **I**nstructions with **T**ask **E**xamples
   A subsystem of SUGILITE *that allows users to use spoken natural language to disambiguate and clarify the intents for their demonstrations.* (Chapter 4)

3. PUMICE
   Programming in a **U**ser-friendly **M**ultimodal **I**nterface through **C**onversations and **E**xamples
   A subsystem of SUGILITE *that allows users to teach conditionals and concepts through a*

[1] The full list of all systems and their acronyms from Brad and his students is available at `https://www.cs.cmu.edu/~bam/acronyms.html`.

mixed-initiative dialog framework. (Chapter 5)

4. EPIDOSITE
   Enabling **P**rogramming of **I**oT **D**evices **O**n **S**martphone **I**nterfaces for **T**he **E**nd-users
   A subsystem of SUGILITE *that allows users to instruct automation rules that involve smart home devices through their corresponding mobile apps.* (Chapter 6)

5. SOVITE
   System for **O**ptimizing **V**oice **I**nterfaces to **T**ackle **E**rrors
   A subsystem of SUGILITE *that allows users to recover from conversational breakdowns caused by natural language understanding errors.* (Chapter 7)

6. PINALITE
   Personal **I**nformation **N**icely **A**nonymized **L**everaging **I**nterface **T**race **E**xamples
   A subsystem of SUGILITE *that allows users to remove personal information from their recorded systems and share their scripts with each other in a privacy-preserving way* (Chapter 8)

7. SCREEN2VEC
   Screen **2**(to) **V**ector **not rock or gemstone themed*
   A self-supervised technique for encoding the semantics of GUI screens into a vector space so screens that are semantically similar to each other can be closer in the space. (Chapter 9)

Chapter 1

Introduction

With the widespread popularity of mobile apps, users are utilizing them to complete a wide variety of tasks ranging from information query (e.g., checking the weather), making purchases (e.g., ordering coffee), communicating (e.g., sending messages), to performing professional job duties (e.g., managing server configurations) [42, 307]. 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 (see Figure 1.1). 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, 66, 241].

However, GUI-based mobile apps have several limitations. First, performing tasks on GUIs can be tedious especially when the task is repetitive. For example, the Starbucks app on Android requires 14 taps to order a cup of venti Iced Cappuccino with skim milk (as of 2019), and even more if the user does not have the account information stored. For such tasks, users would often like to have them automated [6,193,237]. Second, direct manipulation of GUIs is often not feasible or convenient in some contexts, such as when the user is away from the phone or when the user is occupied with other tasks. Third, many tasks require coordination among many apps, but data often remain siloed in individual apps [46]. In a formative study we did with 17 active smartphone users in 2016 (Appendix A), many users reported having to remember information displayed on the GUI of one app, and retype it into the GUI of another app for cross-app tasks. While most smartphone systems support copy-and-paste and many mobile apps utilize the "share to" feature to support cross-app data transfer, these features can be frustrating or difficult to use as reported by users in our interviews. 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 rarely

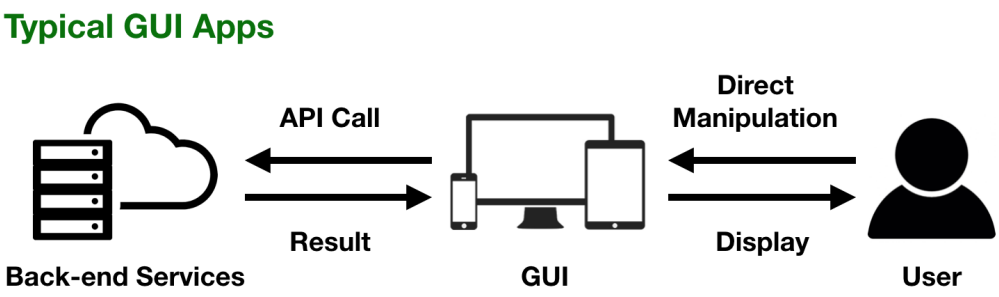


Figure 1.1: The typical role of a GUI. The GUI sits between the user and the back end services, where it receives direct manipulation inputs from the user, sends corresponding calls to the back-end services, and displays the results to the user.

have means to create customized rules or to specify personalized task parameters to reflect their preferences beyond what the app developers have explicitly designed for.

Recently, intelligent agents like Apple Siri, Google Assistant, and Amazon Alexa have become popular solutions to the limitations of GUIs. They can be activated by speech commands to perform tasks on the user's behalf [189]. 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 [38, 252]. 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 hands-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) [185]. 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 (shown in Figure 1.2). 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

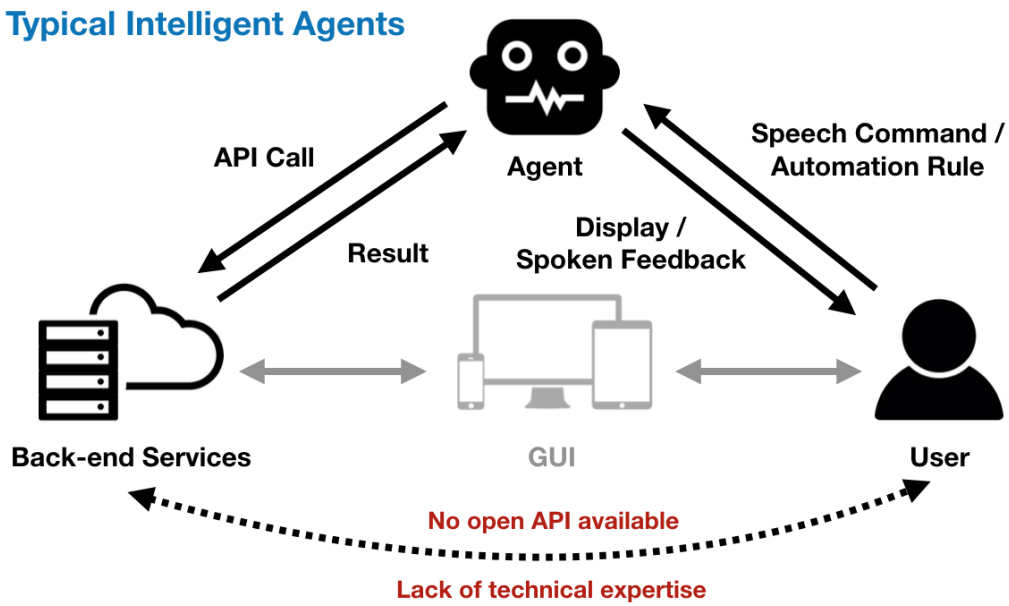


Figure 1.2: The typical role of a prevailing intelligent agent. The agent acts based on the user's speech command or specified automation rule, *directly* invokes back-end services through API calls, and returns the results to the user through either the display or spoken feedback.

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 [293] showed that the usage of these "long-tail" apps made up a significant portion of user app usage. Smartphone users also have highly diverse usage patterns within apps [307] and wish to have more customizability over how agents perform their tasks [64]. 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 the required technical expertise, and *(ii)* the limited availability of openly accessible application programming interfaces (APIs) for many back-end services, as shown in Figure 1.2.

1.1 An End-User Programmable Intelligent Agent

To address this problem, I designed, implemented, and studied a new end-user programmable intelligent agent called SUGILITE[1] [159] in this dissertation. The thesis for my dissertation is:

[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.

A multi-modal end user development system that combines programming by demonstration and natural language instructions can empower users without significant programming expertise to extend and customize intelligent agents for their own app-based computing tasks.

1.1.1 Challenges

Based on prior works in end user development (EUD) for task automation, I identify the following 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., [184,237]) 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., [25, 54]), or services that provide open API access to their functionalities (e.g., [115]). 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 [240] 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 or different parameters. Learning generalized procedures and concepts requires a deeper understanding of 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 [115, 119], while providing great usability due to its simplicity, is not sufficiently expressive for many tasks that users want to automate [273].

- **Robustness:** SUGILITE should be resilient to minor changes in target applications (such as changes in the GUI or menu options), and be able to recover from errors caused by

previously unseen or unexpected situations with the user's help. Macro recorders such as [240] are usually brittle — the macro may break in case of minor changes in the target app GUI. Approaches with complicated programming synthesis or machine learning techniques (e.g., [181, 199]) 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—Conversational breakdowns can decrease users' satisfaction, trust, and willingness to continue using a conversational system and may cause users to abandon the current task [8, 185].

- **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 [152, 155], 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.

1.1.2 My Approach

To address these challenges, SUGILITE takes a *multi-modal* interactive task learning (ITL) [143] 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.

As shown in Figure 1.3, unlike prevailing intelligent agents, SUGILITE sits *between* the user and the GUIs of third-party apps. The user can 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, as shown in Figure 1.2, 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.

This project presents a new human-AI interaction paradigm for interactive task learning. SUGILITE uses existing app GUIs as a *medium* for users to communicate their intents with an AI agent in addition to 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.

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 my approach, SUGILITE *grounds* natural language instructions to demonstration app GUIs, allowing *mutual disambiguation* [221], where demonstrations are used to disambiguate natural language inputs, and vice versa.

In the following chapters, I will go into details of each aspect of SUGILITE's approach and how it addresses the challenges identified in the previous section.

1.2 Contributions

My dissertation contributes a new multi-modal approach for intelligent agents to learn new task procedures and relevant concepts and a system that implements this approach. Specifically, this dissertation makes 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 arbitrary tasks across any or multiple

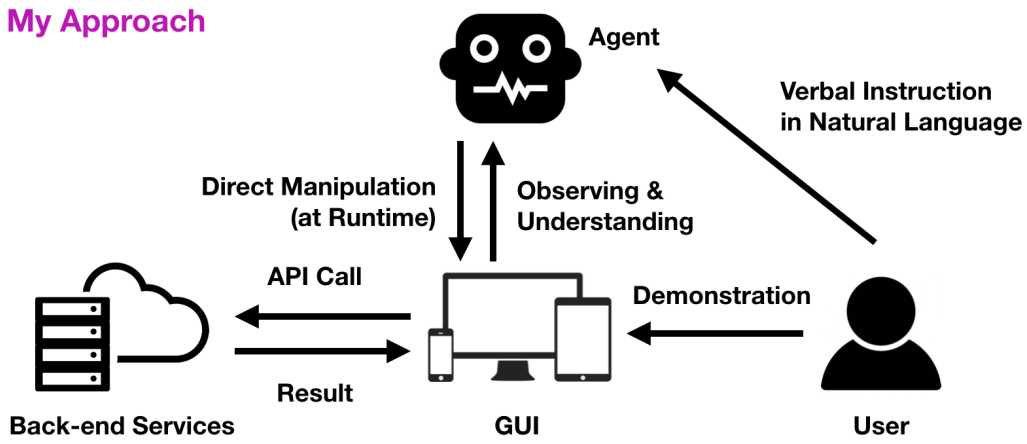


Figure 1.3: The role of the SUGILITE agent *between* the user and the app GUI. At programming time, SUGILITE learns the task procedure and relevant concepts by observing the user's demonstration using the app GUI *while* getting the user's verbal instructions in natural language. At the execution time, SUGILITE directly manipulates the app GUI on the user's behalf to perform the task.

third-party mobile apps through a multi-modal interface combining demonstrations and natural language instructions [159] (Chapter 3).

2. 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 [159] (Chapter 3).

3. 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 [163] (Chapter 4).

4. 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 [163] (Chapter 4).

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 [167] (Chapter 5).

6. An extension of SUGILITE that leverages PBD on mobile apps to control smart home devices through their corresponding mobile app GUIs, addressing the challenge of interoperability in smart home automation by allowing end users to create automation rules across multiple IoT devices from different manufacturers or "eco-systems" [164] (Chapter 6).

7. 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 [161] (Chapter 7).

8. A privacy-preserving sharing mechanism for GUI-based PBD scripts that can identify and obfuscate personal private information embedded in scripts, while maintaining the transparency, readability, robustness, extensibility, and generalizability of original scripts [160] (Chapter 8).

9. A series of lab usability studies that showed end users with different levels of programming expertise were able to successfully use the above systems, mechanisms, and interfaces to complete example tasks derived from common real-world EUD scenarios.

10. A new self-supervised technique for generating semantic embeddings of GUI screens and components that encode their textual contents, visual design, and layout patterns, and app meta-data without requiring manual data annotation [166] (Chapter 9).

1.3 Outline

Chapter 2 describes the background and related work. Chapter 3 covers the design, implementation, and usability evaluation of the core SUGILITE system and its approach in further detail. Chapter 4 discusses the *data description problem* and how SUGILITE addresses this long-standing problem in PBD using an multi-modal approach through its extension APPINITE. Chapter 5 introduces another SUGILITE's extension named PUMICE that allows SUGILITE to learn conceptual knowledge in addition to procedural knowledge about tasks. Chapter 6 talks about the application of SUGILITE's approach in smart home scenarios. Chapter 7 introduces SOVITE, an conversational breakdown repair interface for SUGILITE. Chapter 8 describes the PINALITE system that adds the support for privacy-preserving script sharing among users to SUGILITE. Chapter 9 describes a new

self-supervised technique named SCREEN2VEC for generating semantic representations of GUI screens and components in embedding vectors without requiring manual annotation. Lastly, Chapter 10 discusses the system limitations of SUGILITE, describes some future research directions, and concludes this dissertation.

Chapter 2

Background

This chapter discusses prior work in the following six areas: *(i)* programming by demonstration, *(ii)* natural language programming, *(iii)* multi-modal interfaces, *(iv)* end user development for task automation, *(v)* understanding app interfaces, and *(vi)* breakdown handling in task-oriented dialogues.

2.1 Programming by Demonstration

SUGILITE uses the programming by demonstration (PBD) technique to enable end users to define concepts by referring to the contents 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 [65, 175, 207].

There are many PBD systems that help users automate tasks. However, PBD systems often require access to the internal data of the software where the demonstration happens. This limits the reach of those systems. For example, the CHINLE system [54] only works with interfaces generated with the SUPPLE framework [88], and DocWizards [25] can only record actions performed on SWT widgets in Eclipse. Some PBD systems focus on automating a specific task domain like file manipulation [204], photo manipulation [98], text editing [147], web tasks [16, 152], web scraping [50]), or generating interactive interfaces [85, 194, 276].

A key challenge for PBD is generalization [65,146,175]. When a 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 improves the generalization capability compared with prior similar PBD agents such as CoScripter [152], HILC [120], Sikuli [299], and VASTA [247] through its support for parameterization (Chapter 3), data description disambiguation (Chapter 4), and concept generalization (Chapter 5).

SUGILITE supports domain-independent PBD for task automation by using GUIs of third-party apps. Similar approaches have also been used in prior systems. For example, Assistive Macros [240] uses mobile app GUIs, CoScripter [152], d.mix [110], Vegemite [180], Ringer [16], and PLOW [3] use web interfaces, and HILC [120] and Sikuli [299] use desktop GUIs. Macro recording tools like [240] can record a sequence of input events and replay them later. Many of these tools are quite literal — they replay exactly the same procedure that was demonstrated, with limited ability to generalize the demonstration to perform similar tasks. They are also brittle to any UI changes in the app. Sikuli [299], VASTA [247], and HILC [120] 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.

Compared to those prior systems, SUGILITE can learn task procedures and relevant concepts as generalized knowledge, and create conditionals from natural language instructions. Systems like Sikuli [299] require users to define conditionals in a scripting language, which is not feasible for end users without significant programming expertise.

Another relevant area of research is the use of programming by example (PBE) in data processing and analytics. In these data processing and analytics tools, the user provides multiple input-output examples from which the system creates programs that satisfy the constraints of the examples using program synthesis techniques [106]. One of the most well-known and widely-used PBE applications is the FlashFill in Microsoft Excel, which can analyze the information the user enters and automatically fill data when it identifies a pattern [105]. PBE has also been used in generating labeling rules from interactive demonstrations of users in the interactive data labeling process [79]. A major issue with PBE in data processing and analytics is that end users often have problems coming up with an effective set of examples that are comprehensive, correct, and meaningfully different from each other [151]. The examples provided by users are often ambiguous (i.e., multiple plausible programs can be inferred) [303]. There is also a gap between the user's

mental model of the PBE synthesizer's abilities and the actual abilities of the synthesizer—because the user does not author the programs directly, the user often does not know what the synthesizer can and cannot do [81]. Many interactive PBE systems have been proposed to address these issues, such as [81, 284, 303].

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, 28, 93, 136]. The demonstrations are sometimes also accompanied by natural language instructions [205, 248] similar to SUGILITE. While many recent works try to enhance computing systems' capabilities for parsing human activities (e.g., [235]), modeling human intents (e.g., [91]), and representing knowledge (e.g., [306]) 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 demonstrations but is much easier to comprehend for a computing system.

2.2 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 [12, 26, 176, 200]. In NLP and AI communities, this approach is also known as learning by instruction [10, 55, 142, 178].

The foremost challenge in supporting natural language programming is to deal with the inherent ambiguities and vagueness in natural language [275]. To address this challenge, a prior approach was to require users to use expression styles that resembled conventional programming languages (e.g., [12, 154, 234]), 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 [80] 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 upfront. It also allowed users to form complex commands by combining mul-

tiple 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 [297, 298]. 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 [80], email processing [10, 258], invoking Web APIs [262], or database queries [108, 132, 153].

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 Section 4.3.1) allows SUGILITE to understand user references to GUI contents without having prior knowledge of 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.

2.3 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 [222]. 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 [35]. Prior systems such as CommandSpace [1], Speechify [130], QuickSet [223], SMARTBoard [206], and PixelTone [145] 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., [118, 192]). However, the use of these systems is 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 Chapter 4), the demonstration is the primary modality; verbal instructions are used for disambiguating demonstrated actions. A key pattern used in SUGILITE is *mutual disambiguation* [221]. 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 [190], such as correcting speech recognition errors [263] and assisting the recognition of pen-based handwriting [141]. The recent DoThisHere [296] 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 the approach used in prior work like PLOW [3], which uses users' verbal instructions to infer possible parameters. Compared with PLOW, SUGILITE presents several new interface design features and natural language grounding techniques to further explore how GUI and GUI-based demonstrations can help enhance natural language inputs.

Throughout the design process for SUGILITE, we used user-centered methods [210] including the *natural programming* method [211, 213]. Note that this "natural programming" concept is different from "natural *language* programming" that we discussed before. Natural programming seeks to make the programming process closer to the way the developers think about their tasks [100] to make it easier for developers to implement what they have in mind and to understand the state of their program [211]. My work particularly studies how end users naturally instruct tasks with multiple modalities (see Sections 4.2 and 5.2), and explores the design of multi-modal interfaces that allow users to use the most natural modality for the current context.

2.4 End User Development for Task Automation

Besides programming by demonstration and natural language programming, there are a few other approaches for supporting end user development in task automation. Trigger-action programming (e.g., [92, 115, 119, 274]) has been popular with end users due to its simplicity. In this model, the user can specify a simple pair of trigger and action, where the action will be performed when the trigger happens. Compared with SUGILITE's approach, trigger-action programming tools can only be applied for apps or services that have open APIs available. The simple structure in trigger-action programming also limits its expressiveness so it cannot automate more complex tasks.

Visual programming is another popular technique for end user development. Prior systems like Automate [184] and AppGate [63] allow users to create automation scripts with blocks in flowcharts. Helena [49] also uses a visual language to enable end users to adapt, extend, and understand PBD programs. While visual programming has a lower learning barrier than conventional programming, it still requires users to spend significant effort to learn its syntax, notions, and structures, especially when the programs get more complicated. The supported task domains in visual programming are also often limited by the available "blocks" provided by the system.

While these techniques are much easier to use than conventional programming, the abstractions in visual programming and trigger-action programming still create barriers for end users [92, 113]. To address this issue, IoT Codex [287] uses a paper-engineering tangible approach that conveys and enhances the affordances of the physical world for EUD by allowing users to attached paper RFID-enabled "IoT Stickers" with sensors to everyday objects. SUGILITE seeks to address a similar issue but in different ways. IoT Codex seeks to expand the expressiveness of the physical world for EUD, while SUGILITE uses app GUIs as the medium for users to program automation scripts that can read data from or invoke physical devices.

2.5 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 [72, 73, 74] introduces pixel-based methods to model interactive widgets and interface hierarchies in GUIs, and allowed runtime modifications of widget behaviors. Waken [14] also uses

a computer vision approach to recognize GUI components and activities from screen-captured videos. StateLens [107] and KITE [168] 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 [67] and RICO [66] 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, identifies 23 common flow types (e.g., adding, searching, composing), and can classify the user's GUI interactions into these flow types. Their results show that higher-level semantically meaningful tasks (e.g., search, login) can be recognized from GUI layouts of individual screens based on their design patterns. A similar approach was also used to learn the design semantics of mobile apps, classifying GUI elements into 25 types of GUI components, 197 types of text buttons, and 135 types of icon classes [183]. Appstract [82] focused on the semantic entities (e.g., music, movie, places) instead, extracting entities, their properties, and relevant actions from mobile app GUIs. 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` (Section 9) uses continuous embedding in a vector space for screen representation.

Some prior techniques specifically focus on the visual aspect of GUIs. The RICO dataset [66] 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 [53] and Li et al.'s work [172] 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, our `Screen2Vec` method (Chapter 9) used in SUGILITE provides a more holistic representation of GUI screens by encoding textual contents, GUI component class types, and app-specific meta-data 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 [226] maps the user's natural language commands to target elements on web GUIs. Li et al.'s work [171] goes a step further by generating sequences of actions based on natural language commands. These works use the supervised approach that requires 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 [305] which requires additional developer annotations for the meta-data of GUI components.

Compared with the work on predicting GUI actions, SUGILITE has a slightly different goal for understanding GUI semantics — 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 the sufficient understanding of the contents of the app GUI to be able to handle these verbal instructions in order 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 to understand the user's instruction about the underlying task.

2.6 Breakdown Handling in Task-Oriented Dialogues

A specific focus of this thesis is on handling the conversational breakdowns in SUGILITE's task-oriented dialogues with the users, as discussed in Chapter 7. As reported in Beneteau et al.'s 2019 study [19] of Alexa, conversational breakdown is a major issue experienced by most conversational agent users. The current repair strategies used by users are often not effective since their understandings were frequently inaccurate. Other studies [8, 32, 56, 124, 214, 233] reported similar findings of the types of breakdowns encountered by users and the common repair strategies.

Some previous approaches have been proposed to assist users with discovering, identifying, and repairing conversational breakdowns. A taxonomy of conversational breakdown repair strategies by Ashktorab et al. [8] categorized repair strategies using the 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 make a 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). In the results from that paper [8], 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 this approach requires domain-specific "deep knowledge" about the task and error handling flows manually programmed by the developers [5,197], and therefore is not practical for user-instructed tasks. In fact, even agents with professionally developed conversational skills such as Amazon Alexa and Google Assistant often only provide generic error messages (e.g., "I didn't understand.") with no transparency into the states of understanding in the system and no mechanism for further interactions for repair [56, 233]. In comparison, SUGILITE's approach does not require any additional effort from the developers. It only requires "shallow knowledge" in a domain-general generic language model to map user intents to the corresponding app screens (details in Chapter 7).

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 intent detection

results. This approach is also relevant to work in explainable machine learning (e.g., [261]), which seeks to help users understand the results from intelligent systems and therefore provide more effective inputs. However, these approaches usually require users to verbally clarify or define these keywords. My study (details in Chapter 5) found that when users tried to verbally explain a concept unknown to the system, they often introduced even more unknown concepts in their explanations. The agents also have problems understanding such explanations due to their limited capability of reasoning with natural language instructions and domain knowledge.

My work further explores the design space of improving system transparency—it visualizes the intent detection and slot value extraction results with app GUI screenshots. It also allows the users to easily repair the breakdowns using references to app GUI contents and direct manipulation on the screenshots.

Chapter 3

The SUGILITE System

3.1 Introduction

This chapter describes the design and implementation of the core SUGILITE system, a new intelligent agent that allows end users to teach new tasks and concepts using a *multi-modal* approach that combines *programming by demonstration* (PBD) and learning from natural language instructions. It features the following components:

1. The PBD mechanism that enables users to create automation for arbitrary tasks across any or multiple third-party smartphone apps, and to execute automated tasks through a multi-modal (speech) interface.

2. A PBD script generalization mechanism that leverages the user's verbal commands, demonstrated actions, and the UI hierarchy structures of third-party apps to create generalized programs from single demonstrations. The system also has a representation of the recordings that allows users to manually edit scripts if necessary.

3. An error checking and handling mechanism that improves the system's robustness when the underlying apps change, and also supports further generalization of the scripts to handle new situations as they arise.

Through the work covered in this chapter, we seek to address the challenges of **usability**, **applicability**, and **generalizability**, as identified in Section 1.1.1. The core features described in this

This chapter is modified from the conference paper: Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI 2017)*.

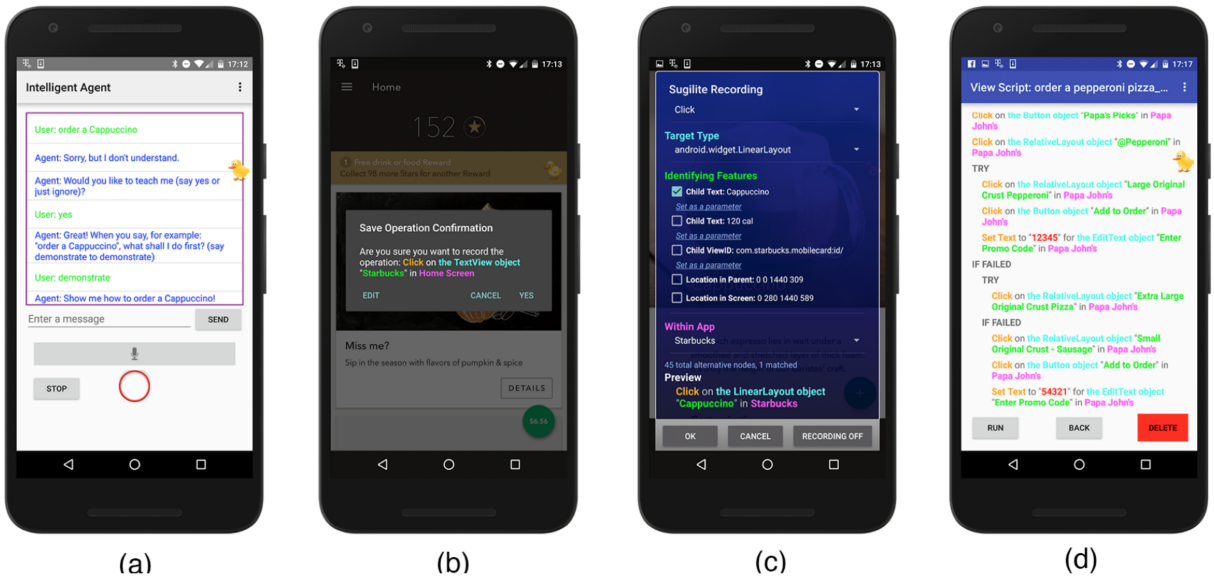


Figure 3.1: Screenshots of the original version of the core SUGILITE system: (a) the conversational interface; (b) the recording confirmation popup; (c) the recording disambiguation/operation editing panel and (d) the viewing/editing script window. *Note that in newer versions, the conversational interface (a) has been replaced with* PUMICE *(Chapter 5), and the data description disambiguation interface (c) has been replaced with* APPINITE *(Chapter 4).*

chapter provide the platform that allows research on the specific issues such as the data description problem (Chapter 4), learning task conditionals and concepts (Chapter 5), and SUGILITE's application in smart home scenarios (Chapter 6). The system described in this chapter is the base version of SUGILITE. Several changes and improvements have been made to this version, which I will cover in the later chapters of this dissertation.

This chapter will also describe a lab usability study that showed that users with different levels of programming experience were able to use SUGILITE to create automation for four tasks derived from real-world scenarios with an 85.5% completion rate, plus subjective feedback showing they liked and would want to use a tool like SUGILITE.

3.2 Example Usage Scenario

This section presents an example scenario of a user interacting with the base version of the SUGILITE system. This example demonstrates SUGILITE's ability to learn generalized tasks with third-party applications from a user's single demonstration.

3.2.1 Order Starbucks Coffee

Smartphone users can order a wide range of products through the apps provided by merchants. In the motivating survey, respondents reported that many such tasks required them to navigate through a complicated multi-level menu in the app to locate the desired offering, which would be especially annoying when the same task is performed frequently. Automating repetitive activities is a key motivation for SUGILITE. In this scenario, we show an example of how a user automates ordering Starbucks drinks using SUGILITE[1].

The user first gives SUGILITE a voice command, "*Order a Cappuccino.*" using the conversational voice interface (Figure 3.1a), for which SUGILITE answers "*Sorry but I don't understand. Would you like to teach me? Say 'demonstrate' to demonstrate.*" The user then says, "*demonstrate*" and starts demonstrating the procedure of ordering a Cappuccino using the Starbucks app.

She first clicks on the Starbucks icon on the home screen, taps on the main menu and chooses "Order", which is exactly the same procedure as what she would do if she is ordering manually through the Starbucks app. (Alternatively, she could instead say verbal commands such as "*Click on Starbucks*", etc.) After each action, a confirmation dialog from SUGILITE pops up (Figure 3.1b) to confirm that the action has been recorded, and which also serves to slow down the user to make sure that the Android accessibility API has time to record the action.

The user continues the "Order" procedure by clicking on "MENU", "Espresso Drinks", "Cappuccinos", "Cappuccino", "Add to Order" and "View Order" in sequence, which are all exactly the same steps that she would do without SUGILITE. In this process, the user could also demonstrate customizing the size, flavor, etc. according to her personal preferences. SUGILITE pops up the confirmation dialog after each click, except for the one on "Cappuccino", where SUGILITE is confused and must ask the user to choose from two identifying features on the same button (explained in Section 3.4): "Cappuccino" and "120cal" (Figure 3.1c). When finished, the user clicks on the SUGILITE status icon and selects "End Recording".

After the demonstration, SUGILITE analyzes the recording and parameterizes the script according to the voice command and its knowledge about the UI hierarchy of the Starbucks app (details in Section 3.4).

This parameterization allows the user to give the voice command "*Order a* `[DRINK]`", where `[DRINK]` can be any of the drinks listed on the Starbucks app's menus. SUGILITE can then order the drink automatically for the user by manipulating the user interface of the Starbucks app.

[1] A demo is available at `https://www.youtube.com/watch?v=KMx7Ea6W6AQ`. This video demo was recorded in 2016.

Alternatively, the automation script can also be executed by using the SUGILITE graphical user interface (GUI) or invoked externally by a third party app using the SUGILITE API.

3.2.2 Additional Examples

To exhibit the generalizability and customizability of SUGILITE, Here are some other example tasks that we have successfully taught the base version of SUGILITE to execute:

1. (American Express) – "Pay off my credit card balance."
2. (Venmo) – "Send `[AMOUNT]` dollars to `[NAME]`."
3. (Fly Delta) – "Find the flights from `[CITY]` to `[CITY]` on `[DATE]`."
4. (CHI 2016) – "Show me all the papers by `[NAME]`."
5. (Transit) – "When is `[BUS LINE]` coming?"
6. (Pokemon Go) – "Collect my coins in the shop."
7. (GrubHub & Uber) – "Request an Uber to the nearest `[TYPE]` restaurant."

3.3 SUGILITE's Key Features

3.3.1 Multi-Modal Interaction

To provide flexibility for users in different contexts, both creating the automation and running the automation can be performed through either the conversational voice interface or SUGILITE's GUI. In order to create an automation script, the user can either give a new voice instruction, for which SUGILITE will reply "... Would you like to teach me?" or the user can start a new demonstration using SUGILITE's GUI.

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

The user can also execute automation scripts by either giving voice commands, or by selecting from a list of scripts. Running an automation script by voice allows the user to give a command

from a distance. For scripts with parameters, the parameter values are either explicitly specified in the GUI or inferred from the verbal command when the conversational interface is used (see Section 3.3.2 for details).

During recording or executing, the user has easy access to the controls of SUGILITE through the floating duck icon (See Figure 3.1, where the icon is on the right edge of the screen). The floating duck icon changes its appearance to indicate the current status of SUGILITE — whether it is recording, executing, or tracking in the background. The user can start, pause or end the execution/recording as well as view the current script (Figure 3.1d) and view the script list from the pop-up menu that appears when the user taps on the duck. The GUI also enables the user to manually edit a script by deleting an operation and all the subsequent operations or to resume recording starting from the end of the script. Selecting an operation lets the user edit it using the editing panel (Figure 3.1c).

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

3.3.2 Script Generalization

Verbal Commands and Demonstrations As shown in the example usage scenario, SUGILITE can automatically identify the parameters in the task and generalize the scripts from a single demonstration. After the user finishes the demonstration, SUGILITE first compares the identifying features of the target UI elements and the arguments of the operations against the verbal command, trying to identify the parameters by matching the words in the command. For example, for the verbal command "find the flights from New York to Los Angeles", SUGILITE identifies "New York" and "Los Angeles" as two parameters if the user typed "New York" into the departure city textbox and "Los Angeles" into the destination textbox during the demonstration.

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

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

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

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

To address this issue, we created a background tracking service that records all the clickable elements in apps and the corresponding previous actions taken to reach each element. This service can run all the time, so SUGILITE can learn about all parts of an app that the user visits. Through this mechanism, we can construct the path to navigate the menu structure to reach a UI element. The text labels of all such elements can then be added to the sets of alternative parameter values for

the scripts. This means that we can allow the user to order drinks that are not an immediate sibling to Cappuccino at the leaf level of the Starbucks order menu tree from a single demonstration.

This method has its trade-offs. First, it brings in false positives. For example, there is a clickable node "Store Locator" in the Starbucks order menu. The generalizing process will then mistakenly add "Store Locator" to the list of what the user can order. Second, running the background tracking affects the phone's performance. Third, SUGILITE cannot generalize for items that were never viewed by the user. Lastly, many participants expressed privacy concerns about allowing background tracking to store text labels from apps, since apps may dynamically generate labels with personal data like an account number or balance. We addressed this problem in the PINALITE subsystem, as discussed in Chapter 8.

3.3.3 Error Checking and Handling

Error checking and handling has been a major challenge for many PBD systems [146]. SUGILITE provides error handling and checking mechanisms to detect when a new situation is encountered during execution or when the app's UI changes after an update. We addressed handing intent recognition errors and entity extraction errors in the SOVITE subsystem, as discussed in Chapter 7.

When executing a script, an error occurs when the next operation in the script cannot be successfully performed. There are many possible reasons for an execution error. First, it is possible that the app has been updated and the layout of the UI has been changed, so SUGILITE cannot find the object specified in the operation. Second, it is possible that the app is currently in a different state than it was during the demonstration. For example, if a user demonstrates how to request an Uber ride during normal pricing, and then uses the script to request a ride during surge pricing, then an error will occur because SUGILITE does not know how to handle the popup from the Uber app that asks for surge price confirmation. Third, the execution can also be interrupted by an external event, like a phone call or an alarm.

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

The "fix the script" option has two sub-options: "replace" and "create a fork", which allow the user to either demonstrate a procedure from the current step that will replace the corresponding part

of the old script, or create a new alternative fork in the old script. The "replace" option should be used to handle permanent changes in the procedure due to an app update or an error in the previous demonstration, or if the user changes her mind about what the script should do. The "create a fork" option (Figure 3.1d) should be used to enable the script to deal with a new situation. The forking works similarly to the try-catch statement in programming languages, where SUGILITE will first attempt to perform the original operation, and then execute the alternative branch if the original operation fails. Allowing SUGILITE to learn other kinds of conditions and the relevant concepts is discussed in Chapter 5.

The forking mechanism can also handle new situations introduced by generalization. For example, after the user demonstrates how to check the score of the NY Giants using the Yahoo! Sports app, SUGILITE generalizes the script so it can check the score of any sports team. However, if the user gives a command "check the score of NY Yankees", everything will work properly until the last step, where SUGILITE cannot find the score because the demonstrations so far only show where to find the score on an American football team's page, which has a different layout than a baseball team's page. In this case, the user creates a new fork and demonstrates how to find the score for a baseball team. Details about how SUGILITE finds items on the screen is described below in Secion 3.4.

3.3.4 Manual Script Editing

For advanced users, we provide an interface to manually edit and manually generalize the script (Figure 3.1d). In this interface, users can delete operations, create forks, or resume recording to add more operations to an existing script. They can also manually generalize the scripts to better reflect their intentions. For example, for the action "Click on 'Chile Mocha Frappuccino in Starbucks 'Featured' menu", the user can override the heuristic-generated identifying feature (see Section 3.4 for details) to use the screen location of the item instead of the text label of the item so the script will click on the top item on the "Featured" menu instead of always choosing the Chile Mocha Frappuccino.

Lastly, the user can manually create parameters. SUGILITE supports the use of a simple markup language. For the action "Set the text of the textbox 'Message' to 'The bus is delayed, I'll be home by 6PM"', the user can manually modify the text to "`@transportation` is delayed, I'll be home by `@time`" so she can reuse the script in a similar scenario by only providing the values for the parameters instead of the whole text. Similarly, the user can set a parameter with the value or label of one UI element (e.g., the result of a search), which can be used for a later operation.

3.4 System Implementation

In this section, we discuss the implementation of the different SUGILITE components and how they are integrated with each other. SUGILITE is an Android application implemented in Java. SUGILITE Scripts and tracking data are stored locally on phone in an SQLite database. SUGILITE does not require jailbreaking/root access to the phone, and has been tested on phones running Android 4.4 through 8.0.

3.4.1 Conversational Interface

The conversational interface we used in the base version of SUGILITE is built upon the Learning by Instruction Agent (LIA)[2] [10]. When the user gives a voice command, the audio is first decoded into text using Google Speech API. LIA uses a Combinatory Categorial Grammar (CCG) parser to parse the verbal command. Based on the parsing result, LIA can initiate a new recording or execute a SUGILITE script. Details on LIA can be found in [10].

3.4.2 Background Accessibility Service

The SUGILITE background service registers as an Android accessibility service. It listens to AccessibilityEvent and distributes the events to the recording handler or the execution handler based on the current mode of the system. The service receives an AccessibilityEvent through the listener whenever a view on the screen is clicked on (or long-clicked on), selected, or focused. The service will also receive an AccessibilityEvent if the text on a view has changed, the state of the current window has changed, or the content of the current window has changed.

3.4.3 Recording Handler

The recording handler in the base version of *Sugilite* receives the AccessibilityEvent from the SUGILITE background accessibility service during the demonstration[3]. It consumes the event if the event is for a user action (click, long click or text entry). From the AccessibilityEvent object, the handler gets meta-data features (text labels, screen locations, accessibility labels, view ID) for the target UI element on which the action was performed. It also includes all the other elements in the hierarchy of the entire current screen (the UI layout), which SUGILITE saves for error checking and

[2] LIA has been replaced with a new semantic parsing mechanism in the latest version, as described in Section 5.4.1.

[3] The recording handler has been updated to use an interaction proxy overlay in the latest version, as described in Section 4.3.3.

generalization. SUGILITE also saves the child features (i.e., element has a child element that...) and the parent features of the target UI element.

SUGILITE applies the following rule-based heuristics to determine a subset of the features to be used in identifying the current screen and the current target UI element. The SUGILITE first looks for the text or accessibility label in the UI element. If it has one, the system will verify that this feature is unique among all the UI elements on the screen. If this fails, The SUGILITE seeks to find a unique second-level feature like view ID or label of a child element. If The SUGILITE fails to find a feature to uniquely identify the element on the screen, it will use the screen coordinates of the element's bounding box. The chosen subset of features needs to *(i)* uniquely identify the UI layout of the screen and the specific target UI element on that screen, and *(ii)* still work for future runs of the application to identify the same element.

If the recording handler determines that the heuristic-generated feature set can fulfill the above two requirements, the confirmation popup (Figure 3.1b) will be shown. Otherwise, the disambiguation panel (Figure 3.1c) will be shown to ask the user to manually disambiguate what features should be used in the script[4]. After this, The SUGILITE generates and saves an operation to the current recorded script. The operation also stores identifiers for all the unique UI layouts and all the alternative UI elements that are structurally in parallel with the target UI element. After the demonstration, SUGILITE will use these when parameterizing and generalizing the script, as described earlier.

3.4.4 Execution Handler

The execution handler will be activated when an AccessibilityEvent is sent from the background accessibility service, which happens when the screen changes, or when the user takes any action. The execution handler will try to match the current UI layout to the stored one in the script, then it tries to find the required target UI element on the screen, and then it will try to perform the operation as specified in the current operation. In the SUGILITE UI, the floating icon of a duck will also move to the target UI element to signify the operation. If the current screen specified in the AccessibilityEvent was not among the recorded ones for the current operation, the error handling pop-up will be shown, signifying that the state of the current app does not match what was demonstrated for the operation.

If the current operation is successful, the execution handler will then proceed to handle the next operation. The error handling mechanism (described earlier) will be activated in case of an error.

[4]The manual disambiguation interface has been replaced with a multi-modal disambiguation interface, as discussed in Chapter 4.

3.4.5 SUGILITE API

SUGILITE has an API that allows external apps to utilize its recording, tracking, and executing functionality. Scripts can be exported for editing/sharing or imported for executing in JSON format. Several research software projects such as PrivacyStream [170] and the Yahoo InMind[5] Middleware was building upon, connected to, or invoking SUGILITE.

3.5 User Study

To assess how successfully users with various levels of prior programming experience can use some features of SUGILITE, we conducted a lab study where we asked the participants to teach SUGILITE new tasks for four given scenarios.

3.5.1 Participants

19 participants aged 20–30 (mean = 24.2, SD = 2.55) were recruited from the local university community. The participants were required to be 18 or order, be active smartphone users and be fluent in English. All recruited participants were university students.

In the recruiting form, participants rated their own programming experience on a five-point scale from "no experience" to "experienced programmer". We specifically selected participants across a range of prior programming experience. Table 3.1 shows the description used for each category and the number of participants in each category.

Group–Programming Experience	#
1 – I've never done any computer programming	3
2 – I've done some light programming (e.g., Office macros, excel functions, simple scripts)	4
3 – Beginner programmer (experience equivalent to 1–2 college level computer science classes)	5
4 – Intermediate programmer (1–2 years of programming experience)	3
5 – Experienced programmer (2+ years of programming experience)	4

Table 3.1: Number of participants (#) grouped by programming experience

[5] https://www.cmu.edu/homepage/computing/2014/winter/project-inmind.shtml

3.5.2 Sample Tasks

We chose five tasks for the study based on the common repetitive task scenarios from a motivating survey. The first task was used as a tutorial, where the experimenter showed the participant how to teach SUGILITE to complete the example task and explained how to operate SUGILITE. The other four tasks were given to the participants in random order. All the participants used the same Nexus 6 phone with SUGILITE and relevant apps installed.

Before each task, the participants were given time to get familiar with the involved apps (Uber, Starbucks, Yahoo! Sports, and Gmail), so they were all proficient at performing the tasks using direct manipulation. We first asked the participants to perform the task directly without SUGILITE, and then asked them to teach SUGILITE the same task by demonstration. During the task, we would not answer questions on how to use SUGILITE (but we responded to the requests for clarification on the task specifications).

Below are the task descriptions:

Tutorial Task: Pizza Ordering In this task, we demonstrated the procedure of ordering a large pepperoni pizza for carryout at the nearest Papa John's store using the Papa John's app. The script was then automatically generalized so it can be used to order any of the three basic variations of pizza (pepperoni, cheese, and sausage). There were one `SET_TEXT` and nine `CLICK` operations in the script. Among these operations, two `CLICK` operations required manual disambiguation of the identifying features.

Task 1: Get an Uber The specification given to the participants was "*You should teach the agent how to use the Uber app to request an Uber X cab to the current location.*" The standard procedure for this task had two `CLICK` operations, none of which required manual disambiguation of the identifying features. The participants were told to not confirm sending the Uber request to avoid being charged.

Task 2: Check Sports Score The specification given to the participants was "*You should teach the agent how to use the Yahoo! Sports app to show you the latest score of Pittsburgh Steelers.*" The standard procedure for this task had four `CLICK` operations and one `SET_TEXT` operation, none of which required manual disambiguation of the identifying features. The parameter for the `SET_TEXT` operation is automatically generalized so the script can be used to check the score for any football team in Yahoo! Sports.

Task 3: Order Coffee The specification given to the participants was "*You should teach the agent how to use the Starbucks app to order a cup of Cappuccino.*" The standard procedure for this task had 11 `CLICK` operations. Among them, one `CLICK` operation required manual disambiguation of the identifying features. This script is automatically generalized so it can be used to order any drink from the Starbucks app. The participants were told to not submit the final order to avoid being charged.

Task 4: Send an Email The specification given to the participants was "*You should teach the agent the command 'Tell Joe that I will be late because my car is broken.' by demonstrating how to send a new email to 'joe@example.com', with the subject 'I will be late' and body 'I will be late because my car is broken'.*" The standard procedure for this task had four `CLICK` operations and three `SET_TEXT` operations, none of which required manual disambiguation of the identifying features. This script is automatically generalized so it can respond to "Tell `[NAME]` that `[SOMETHING]` because `[SOMETHING]`."

3.5.3 Procedure

The study took about 1 hour per participant. After signing the consent form, the participant received the tutorial through the experimenter walking through the tutorial task. The tutorial took about 5 minutes. Following the tutorial, the experimenter gave the first task to the participant. Then the participant began to do the tasks as specified in the previous section. After performing the tasks, the participant filled out a usability questionnaire on their experiences interacting with SUGILITE. Finally, the experimenter conducted a brief semi-structured interview with the participant based on the questionnaire responses and the experimenter's observations during the study. Participants were compensated $15 for their time.

3.5.4 Results

Overall, 65 out of 76 (85.5%) scripts created by the participants ran and performed the intended task successfully. 8 out of the 19 (42.1%) participants succeed in all four tasks. 10 (52.6%) succeeded in three tasks and 1 (5.3%) succeeded in only two tasks. All participants completed at least two tasks successfully. A Pearson's chi-squared test was performed and *no* significant relationship was found between task completion and the level of prior programming experience ($X^2(4) = 4.15$, $p = 0.39$).

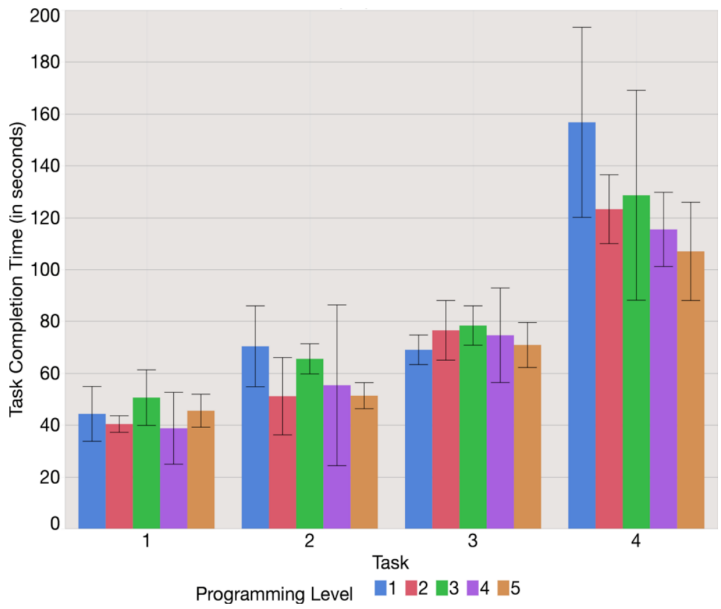


Figure 3.2: The average task completion time for the participants grouped by their programming experience. Shorter bars are better. Error bars show the standard deviations.

For each successful task, we measured the task completion time from when the voice command was successfully received until the participant ended the recording. We then used a one-way ANOVA to compare the task completion time of each task for participants grouped by their programming experience. No significant difference between the five groups as described in Table 1 based on task completion time was found for any of the tasks. The average task completion time of each task by group is shown in Figure 3.2. All times are in seconds.

Task 1 All but one participant completed this task (94.7%). That participant "double clicked" (i.e., clicked twice in a row without waiting for the confirmation pop-up in between), which caused the system to fail to record the first action. This issue has been fixed in the subsequent versions of SUGILITE.

Task 2 15 participants out of 19 (78.9%) completed this task. Four participants failed by entering the text directly into textboxes[6] (they were supposed to double tap on the textbox and type into a pop-up due to an implementation limitation).

[6]The latest version of SUGILITE has fixed this issue by adding support for direct text input.

Task 3 17 participants out of 19 (89.5%) completed this task. A participant erroneously "double clicked" and another participant recorded a click when the Starbucks app had not finished loading, which would cause the execution to block when this click is to be performed.

Task 4 14 participants out of 19 (73.7%) completed this task. Five participants failed by entering the text directly into textboxes.

3.5.5 Time Trade-off

For each of the four tasks, we also calculated an average "break-even" point n, for which if a task needs to be performed for at least n times, then the total time needed for automating the task and executing the script for n times is shorter than the time needed to do the tasks manually for n times. This metric is also known as the *equivalent task size* [203]. We use n as a rough measure for the time trade-off of using SUGILITE to automate tasks. In plain words, if a user needs to perform a task more than n times, then automating the task with SUGILITE can save her time.

Using the average time it took to create the automation ($\bar{t}$), the average time it took to perform the task manually ($\bar{t_m}$) and the time it took to execute the automation ($\bar{t_0}$), we calculate the average break-even value of n for each task, shown in Table 3.2.

Task	$\bar{t}$	$\bar{t_m}$	$\bar{t_0}$	n
Task 1	44.47s	13.71s	3.35s	5
Task 2	58.61s	15.15s	5.12s	6
Task 3	74.29s	26.47s	7.34s	4
Task 4	125.25s	50.25s	6.14s	3

Table 3.2: Average time to automate the task ($\bar{t}$), average time to perform the task manually ($\bar{t_m}$), time to run the automation ($\bar{t_0}$), and the "break-even" point (n) for the four tasks.

3.5.6 Subjective Feedback

Overall, SUGILITE received positive feedback on both usability and usefulness from our study participants. On the post questionnaire, the participants were asked to rate their agreement with statements related to their experience interacting with SUGILITE on a 7-point Likert scale from "Strongly Disagree" to "Strongly Agree." Table 3.3 shows the average score for the usability-

related items on the questionnaire. Table 3.4 shows the average score for the usefulness-related questions.

Statement	Score
"It's easy to learn how to use this system."	6.17
"My interaction with the system is clear and understandable."	6.00
"I'm satisfied with my experience using this system."	5.94

Table 3.3: Average scores on usability questions from the post-questionnaire (on a 7-point scale).

Statement	Score
"I find the system useful in helping me creating automation."	6.39
"I find automating tasks with the system is efficient."	6.11
"I would use this system to automate my tasks."	6.06

Table 3.4: Average scores on usefulness questions from the post-questionnaire (on a 7-point scale).

In a semi-structured interview after the questionnaire, the participants were asked whether they found anything unclear or confusing in their interaction with SUGILITE. Most complaints were on the demonstration of text entry, where the user needs to type into a SUGILITE popup instead of the textbox in the original app. Participants found this to be unnatural and easy-to-forget[7]. Some also reported information overload on the disambiguation panel (Figure 1c). It contained too much information, which made it hard to locate where they needed to read and make selections[8].

3.5.7 Discussion

The outcome of the evaluation suggests no significant difference based on the level of programming experience of participants for all four tasks in either task completion rate or task completion time. The groups with no programming experience (Group 1) and only light programming experience (Group 2) completed 25 out of 28 (89.3%) tasks. The results indicate that end users with little or no programming experience can successfully use SUGILITE to automate smartphone tasks.

[7] As mentioned above, the latest version of SUGILITE has added the support for direct text input.

[8] The manual disambiguation has been replaced with a multi-modal disambiguation interface, as discussed in Chapter 4.

Looking at the "break-even" point for each task, we learn that for the four example scenarios, the user can save time with SUGILITE if they are to perform the tasks more than 3 to 6 times. This implies that the efficiencies of many repetitive tasks could potentially benefit from automating through SUGILITE, because the overhead of creating automation using SUGILITE is small.

After the study, we asked the participants to describe scenarios from their own smartphone usage where SUGILITE would be helpful. Some of the scenarios involved using apps that are not likely to ever be integrated into existing agents, like the specialized apps made for the university community to check the shuttle location, dining menu, meeting room availability, etc. Many organizations or communities have made such apps to serve the information needs of their members. Due to the limited engineering resources available and the small user base for those apps, they are unlikely to get integrated into the intelligent software agents for automation without EUD. But for the users of those apps, they often use the apps frequently and repetitively, so they wish to have their common tasks automated.

The participants commented that even when apps have already been integrated into the prevailing agents like Siri, sometimes specific tasks they wanted are not supported. An example is that for the music player, users cannot change the equalizer settings or download the current song using voice commands. They also wished to incorporate personalized settings into the automation (e.g., setting the volume before playing a particular song).

They also proposed scenarios for creating SUGILITE automation beyond single apps. Users often perform a set of tasks in a row (e.g., check the weather, the traffic information and book a cab when waking up) so Sugilite can be used to create a single voice command for multiple tasks. Another example is a command "request an Uber to the nearest (sushi restaurant, pharmacy, grocery store...)", SUGILITE can first use Google Maps to retrieve the address of the nearest desired entity, then use the Uber app to request a cab with the destination filled in.

3.6 Chapter Conclusion

This chapter described the design and implementation of the core SUGILITE system, an end-user-programmable intelligent agent that can learn tasks in various task domains from end users through demonstrations on existing third-party app GUIs, and infer parameters and their associated possible values in learned tasks from a combination of task demonstrations, user verbal instructions, and app GUI hierarchy structures.

Even though there have been many years of research on programming by demonstration and speech interfaces, SUGILITE was the first PBD system to show how they can successfully be put together on a smartphone to enhance the capabilities of both [159].

Chapter 4

The Data Description Problem

4.1 Introduction

A central challenge for PBD is generalization. The PBD system should produce more than literal record-and-replay macros (e.g., sequences of keystrokes and clicks at specific screen locations), but learn the target task at a higher level of abstraction so it can perform similar tasks in new contexts [65, 175]. Previously, Section 3.3.2 discussed parameterization, which is a key issue in generalization. This chapter will cover another key issue in generalization — the *data description problem* [65, 175]: when the user performs an action on an item in the GUI, what does it mean? The action and the item have many features. The system needs to choose a subset of features to describe the action and the item, so that it can correctly perform the right action on the right item in a different context.

For example, in Figure 4.1(a), the user's action is "Click", and the target object can be described in many different ways, such as Charlie Palmer Steak / the second item from the list / the closest restaurant in Midtown East / the cheapest steakhouse, etc. The system would need to choose a description that reflects the user's intention, so that the correct action can be performed if the script is run with different search results.

To identify the correct data description, prior PBD systems have varied widely in the division of labor, from making no inference and requiring the user to manually specify the features, to using sophisticated AI algorithms to automatically induce a generalized program [212]. Some prior systems such as SmallStar [109] and Topaz [209] used the "no inference" approach to give

This chapter is modified from the conference paper: Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M. Mitchell, and Brad A. Myers. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2018).*

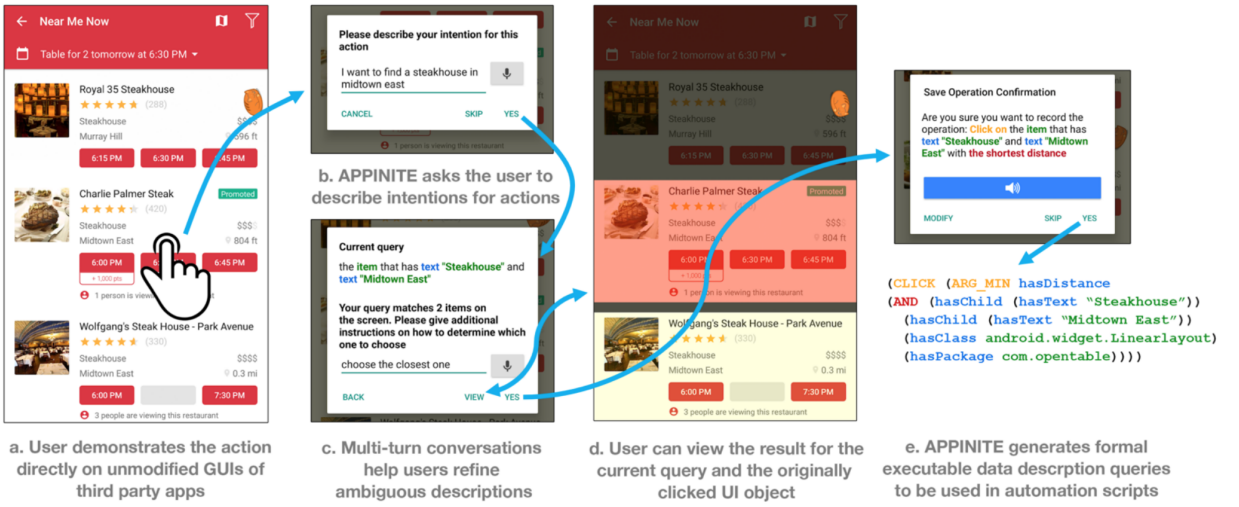


Figure 4.1: Specifying data description in programming by demonstration using APPINITE: (a, b) enable users to naturally express their intentions for demonstrated actions verbally; (c) guides users to formulate data descriptions to uniquely identify target GUI objects; (d) shows users real-time updated results of current queries on an interaction overlay; and (e) formulates executable queries from natural language instructions.

users full control in manually choosing features to use. However, this approach involves heavy user effort, and has a steep learning curve, especially for end users with little programming expertise. Others like Peridot [208] and CoScipter [152] went a step further and used heuristic rules for generalization, which were still limited in applicability. This approach can only handle simple scenarios (unlike Figure 4.1), and has the possibility of making incorrect assumptions.

At the other end of the spectrum, prior systems such as [104, 148, 195, 199] used more sophisticated AI-based programming synthesis techniques to automatically infer the generalization, usually from multiple example demonstrations of a task. However, this approach has issues as well. It requires a large number of examples, but users are unlikely to be willing to provide more than a few examples, which limits the feasibility of this approach [146]. Even if end users provide a sufficient number of examples, prior studies [151, 212] have shown that untrained users are not good at providing useful examples that are meaningfully different from each other to help with inferring data descriptions. Furthermore, users have little control of the resulting programs in these systems. The results are often represented in such a way that is difficult for users to understand. Thus, users cannot verify the correctness of the program, or make changes to the system [146], resulting in a lack of trust, transparency and user control.

This chapter describes a new multi-modal interface for SUGILITE named APPINITE[1]. This interface enables end users to naturally express their intentions for data descriptions when programming task automation scripts by using a combination of demonstrations and natural language instructions on the GUIs of arbitrary third-party mobile apps. APPINITE helps users address the data description problem by guiding them to verbally reveal their intentions for demonstrated actions through multi-turn conversations. APPINITE constructs data descriptions of the demonstrated action from natural language explanations. This interface is enabled by our novel method of constructing a semantic relational knowledge graph (i.e., an ontology) from a hierarchical GUI structure (e.g., a DOM tree). APPINITE uses an interaction proxy overlay to highlight ambiguous references on the screen, and to support meta actions for programming with interactive UI widgets in third-party apps. The APPINITE interface replaces the manual disambiguation panel (described in Section 3.4 and shown in Figure 3.1c) in the base SUGILITE system.

APPINITE provides users with greater expressive power to create flexible programming logic using the data descriptions, while retaining a low learning barrier and high understandability for users. The evaluation showed that APPINITE is easy-to-use and effective in tasks with ambiguous actions that are otherwise difficult or impossible to express in prior PBD systems.

4.2 Formative Study

We conducted a formative study to understand how end users may verbally instruct the system simultaneously while demonstrating using the GUIs of mobile apps, and whether these instructions would be useful for addressing the data description problem. We asked workers from Amazon Mechanical Turk (mostly non-programmers [116]) to perform a sample set of tasks using a simulated phone interface in the browser, and to describe the intentions for their actions in natural language. We recruited 45 participants, and had them each perform 4 different tasks. We randomly divided the participants into two groups. One group of participants were simply told to narrate their demonstrations in a way that would be helpful even if the exact data in the app changed in the future. Another group were additionally given detailed instructions and examples of how to write good explanations to facilitate generalization from demonstrations.

After removing responses that were completely irrelevant, or apparently due to laziness (32% of the total), the majority (88%) of descriptions from the group that were not given detailed instructions and all of descriptions (100%) from the group that received detailed instructions explained in-

[1] APPINITE is named after a type of rock. The acronym stands for **A**utomation **P**rogramming on **P**hone **I**nterfaces using **N**atural-language **I**nstructions with **T**ask **E**xamples.

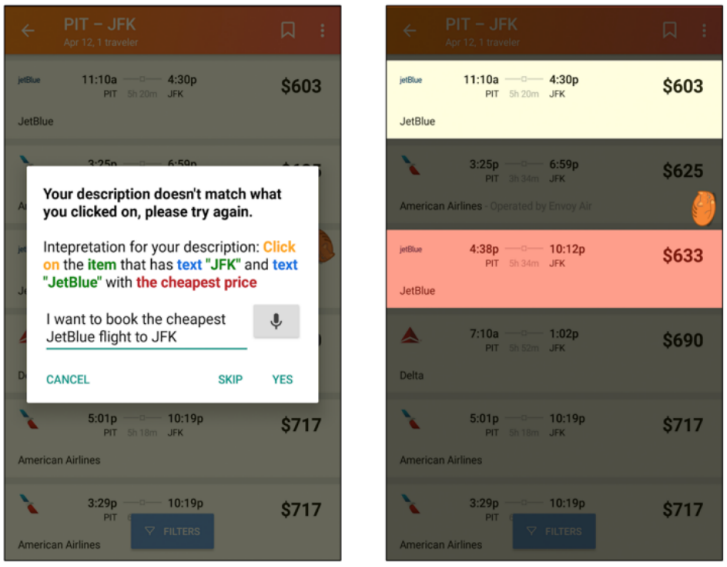
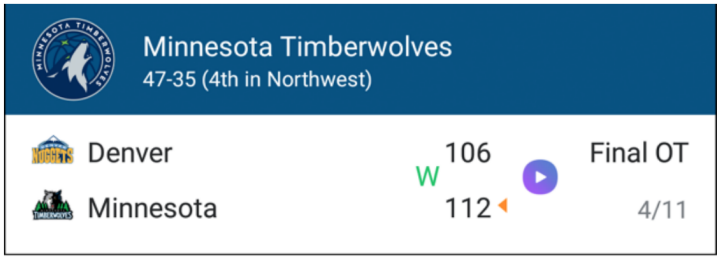


Figure 4.2: APPINITE's error handling interfaces for handling situations where the instruction and the demonstration do not match.

tentions for the demonstrations in ways that would facilitate generalization, e.g., by saying "*Scroll through to find and select the* ***highest rated action film****, which is Dunkirk*" rather than just "*select Dunkirk*" without explaining the characteristic feature behind their choice.

We also found that many of such instructions contain spatial relations that are either explicit (e.g., "*then you click the back button* ***on the bottom left***") or implicit (e.g., "*the reserve button* ***for*** *the hotel*", which can translate to "the button with the text label 'reserve' that is **next to** the item representing the hotel"). Furthermore, approximately 18% of all 1631 natural language statements we collected from this formative study used some generalizations (e.g., the highest rated film) in the data description instead of using constant values of string labels for referring to the target GUI objects. These findings illustrate the need for constructing an intermediate level representation of GUIs that abstracts the semantics and relationships from the platform-specific implementation of GUIs and maps more naturally to the semantics of likely natural language explanations. We addressed this need through the extraction of UI Snapshot Knowledge Graph, as described in Section 4.3.1.




```
(DEFINE score (AND (isNumeric true)
  (hasClass android.widget.TextView)
  (rightTo (AND (hasText "Minnesota")
                (hasClass android.widget.TextView)))))
```

Figure 4.3: A snippet of an example GUI where the alignment suggests a semantic relationship — "*This is the score for Minnesota*" translates into "Score is the `TextView` object with a numeric string that is to the right of another `TextView` object Minnesota."

4.3 APPINITE's Design and Implementation

In this section, we discuss the design and implementation of three core components of APPINITE: the UI snapshot graph extractor, the natural language instruction parser, and the interaction proxy overlay.

4.3.1 UI Snapshot Knowledge Graph Extraction

We found in the formative study that end users often refer to spatial and semantic-based generalizations when describing their intentions for demonstrated actions on GUIs. Our goal is to translate these natural language instructions into formal executable queries of data descriptions that can be used to perform these actions when the script is later executed. Such queries should be able to generalize across different contexts and small variations in the GUI to still correctly reflect the user's intentions. To achieve this goal, a prerequisite is a representation of the GUI objects with their properties and relationships, so that queries can be formulated based on this representation.

APPINITE extracts GUI elements using the Android Accessibility Service, which provides the content of each window in the current GUI through a static hierarchical tree representation similar to the DOM tree used in HTML. Each node in the tree is a *view*, representing a UI object that is visible (e.g., buttons, text views, images) or invisible (often created for layout purposes). Each view also contains properties such as its Java class name, app package name, coordinates for its on-screen bounding box, accessibility label (if any), and raw text string (if any). Unlike a DOM, our

extracted hierarchical tree does not contain specifications for the GUI layout other than absolute coordinates at the time of extraction. It does not contain any programming logic or meta-data for the text values in views, but only raw strings from the presentation layer. This hierarchical model is not adequate for APPINITE's data description, as it is organized by parent-child structures tied to the implementation details of the GUI, which are invisible to end users of the PBD system. The hierarchical model also does not capture geometric (e.g., next to, above), shared property value (e.g., two views with the same text), or semantic (e.g., the *cheapest* option) relations among views, which are often used in users' data descriptions.

To represent and to execute queries used in data descriptions, APPINITE constructs relational knowledge graphs (i.e., ontologies) from hierarchical GUI structures as the medium-level representations for GUIs. These *UI snapshot graphs* abstract the semantics (values and relations) of GUIs from their platform-specific implementations, while being sufficiently aligned with the semantics of users' natural language instructions. Figure 4.4 illustrates a simplified example of a 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 [34], Wikidata [282] and WikiBrain [162], 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 property 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 Figure 4.3 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.

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.

4.3.2 Instruction Parsing

After APPINITE constructs a UI snapshot graph, the next step is to parse the user's natural language description into a formal executable query to describe this action and its target UI object. In APPINITE, we represent queries in a simple but flexible LISP-like query language (*S*-expressions) that can represent joins, conjunctions, superlatives and their compositions. Figure 4.1e, Figure 4.4, and Figure 4.3 show some example queries. To support these queries, we implemented a new semantic parser that replaces the old LIA parser used in the initial version of SUGILITE (see Section 3.4).

Representing UI elements as a UI snapshot graph offers a convenient data abstraction model for formulating a query using language that is closely aligned with the semantics of users' instructions during a demonstration. For example, the utterance "*next to the button*" expresses a natural *join* over a binary relation near and a unary relation `isButton` (a unary relation is a mapping from all UI object entities to truth values, and thus represents a subset of UI object entities.) An utterance "*a textbox next to the button*" expresses a natural *conjunction* of two unary relations, i.e., an intersection of a set of UI object entities. An utterance such as "*the cheapest flight*" is naturally expressed as a *superlative* (a function that operates on a set of UI object entities and returns a single entity, e.g., `ARG_MIN` or `ARG_MAX`). Formally, we define a data description query in AP-

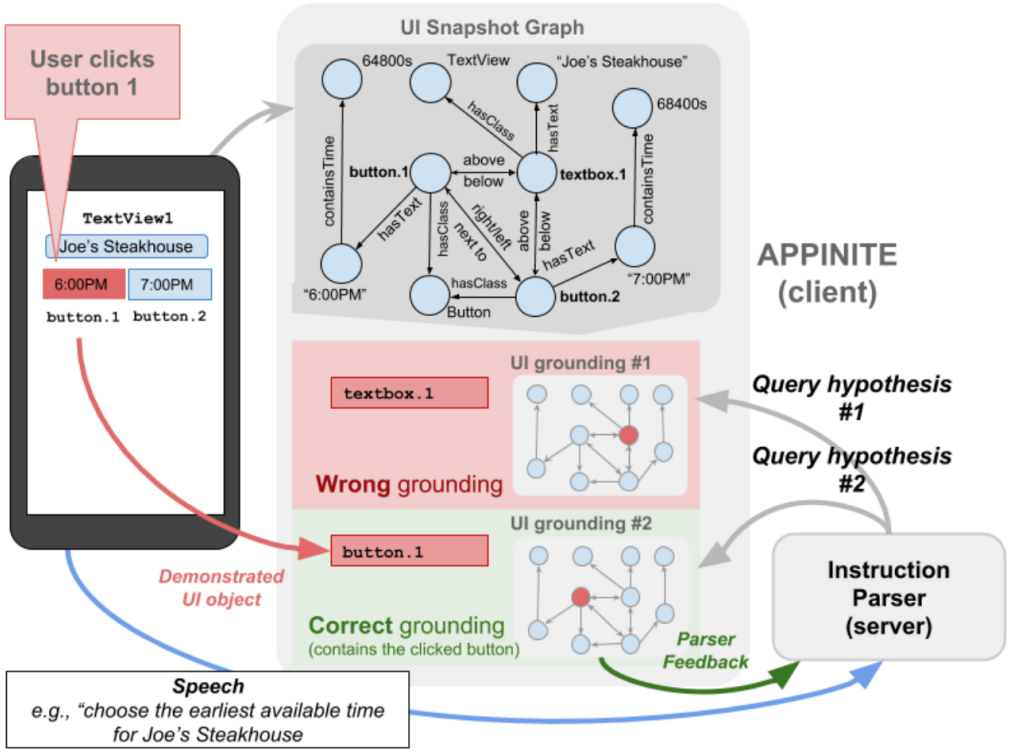


Figure 4.4: APPINITE's instruction parsing process illustrated on an example UI snapshot graph constructed from a simplified GUI snippet.

PINITE's language as an *S*-expression that is composed of expressions that can be of three types: *joins*, *conjunctions* and *superlatives*, 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. APPINITE's language forms a subset of a more general formalism known as Lambda Dependency-based Compositional Semantics [173], a notationally simpler alternative to lambda calculus, which is particularly well-suited for expressing queries over knowledge-graphs.

APPINITE's semantic parser uses a Floating Parser architecture [227] and does not require hand-engineering of lexicalized rules, e.g., as is common with synchronous CFG or CCG based semantic parsers. This allows users to express lexically and syntactically diverse, but semantically equivalent statements such as "*I am going to choose the item that says coffee with the lowest price*" and "*click on the cheapest coffee*" without requiring the developer to hand-engineer or tune the grammar for different apps. Instead, the parser learns to associate lexical and syntactic patterns (e.g., associating the word "cheapest" with predicates `ARG_MIN` and `containsPrice`) with semantics during training via rich features that encode co-occurrence of unigrams, bigrams and skipgrams with predicates and argument structures that appear in the logical form. We trained the parser used in the preliminary usability study via a small number of example utterances paired with annotated logical forms and knowledge-graphs (840 examples), using 4 of the 8 apps used in the user studies as a basis for training examples. We use the core Floating Parser implementation within the SEMPRE framework [24].

4.3.3 Interaction Proxy Overlay

The base version of SUGILITE instruments GUIs by passively listening for the user's actions through the Android accessibility service, and popping up a disambiguation dialog *after* an action if clarification of the data description is needed (see Section 3.4). This approach allows PBD on unmodified third-party apps without access to their internal data, which is constrained by working with Android apps (unlike web pages, where run-time interface modification is possible [33, 75, 271]). However, at the time when the dialog shows up, the context of the underlying app may have already changed as a result of the action, making it difficult for users to refer back to the previous context to specify the data description for the action. For example, after the user

taps on a restaurant, the screen changes to the next step, and the choice of restaurant is no longer visible. This approach also has problems processing non-standard GUIs whose widgets do not use the standard `onClick()` listeners for handling clicks.

To address these issues, we implemented an interaction proxy [304] in APPINITE to add an interactive overlay on top of third-party GUIs. This mechanism replaced the old recording handler discussed in Section 3.4. It can run on any phone running Android 6.0 or above, without requiring root access. The full-screen overlay can intercept all touch events (including gestures) before deciding whether, or when to send them to the underlying app, allowing APPINITE to engage in the disambiguation process while preventing the demonstrated action from switching the app away from the current context. Users can refine data descriptions through multi-turn conversations, try out different natural language instructions, and review the state of the underlying app when demonstrating an action without invoking the action.

The overlay is also used for conveying the state of APPINITE in the mixed-initiative disambiguation to improve transparency. An interactive visualization highlights the target UI object in the demonstration, and matched UI objects in the natural language instruction when the user's instruction matches multiple UI objects (Figure 4.1d), or the wrong object (Figure 4.2a). This helps users to focus on the differences between the highlighted objects of confusion, assisting them to come up with additional differentiating criteria in follow-up instructions to further refine data descriptions. In the "verbal-first" mode where no demonstration grounding is available, APPINITE also uses similar overlay highlighting to allow users to preview the matched object results for the current data description query on the underlying app GUI.

4.4 User Study

We conducted a lab usability study. Participants were asked to use APPINITE to specify data descriptions in 20 example scenarios. The purpose of the study was to evaluate the usability of APPINITE on combining natural language instructions and demonstrations.

4.4.1 Participants

We recruited 6 participants (1 identified as female and 5 identified as male, average age = 26.2) at Carnegie Mellon University. All but one of the participants were graduate students in technical fields. All participants were active smartphone users, but none had used APPINITE or SUGILITE prior to the study. Each participant was paid $15 for an 1-hour user study session.

Although the programming literacy of our participants is not representative of APPINITE's target users, this was not a goal of this study. The primary goal was to evaluate the usability of APPINITE's interaction design on combining natural language instructions and demonstrations. The demonstration part of this usability study was based on the earlier SUGILITE study (see Section 3.5), which found no significant difference in PBD task performances among groups with different programming expertise. The formative study (Section 4.2) showed that non-programmers were able to provide adequate natural language instructions from which APPINITE can generate generalizable data descriptions.

4.4.2 Tasks

From the top free apps in Google Play, we picked 8 sample apps (OpenTable, Kayak, Amtrak, Walmart, Hotel Tonight, Fly Delta, Trulia and Airbnb) where we identified data description challenges. Within these apps, we designed 20 scenarios. Each scenario required the participant to demonstrate choosing an UI object from a collection of options. All the target UI objects had multiple possible and reasonable data descriptions where the correct ones (that reflect user intentions) could not be inferred from demonstrations alone, or using heuristic rules without semantic understanding of the context. The tasks required participants to specify data descriptions using APPINITE. For each scenario, the intended feature for the data description was communicated to the participant by pointing at the feature on the screen. Spoken instructions from the experimenter were minimized, and carefully chosen to avoid biasing what the participant would say. Four out of the 20 scenarios were set up in a way that multi-turn conversations for disambiguation (e.g., Figure 4.1c and Figure 4.1d) were needed. The chosen sample scenarios used a variety of different domains, GUI layouts, data description features, and types of expressions in target queries (i.e. joins, conjunctions and superlatives).

4.4.3 Procedure

After obtaining consent, the experimenter first gave each participant a 5-minute tutorial on how to use APPINITE. During the tutorial, the experimenter showed a video[2] as an example to explain APPINITE's features.

Following the tutorial, each participant was shown the 8 sample apps in random order on a Nexus 5X phone. For each scenario within each app, the experimenter navigated the app to the designated state before handing the phone to the participant. The experimenter pointed to the UI

[2] https://www.youtube.com/watch?v=2GqMUiPvidU

object which the participant should demonstrate clicking on, and pointed to the on-screen feature which the participant should use for verbally describing the intention. For each scenario, the participant was asked to demonstrate the action, provide the natural language description of intention, and complete the disambiguation conversation if prompted by APPINITE. The participant could retry if the speech recognition was incorrect, and try a different instruction if the parsing result was different from expected. APPINITE recorded participants' instructions as well as the corresponding UI snapshot graphs, the demonstrations, and the parsing results.

After completing the tasks, each participant was asked to complete a short survey, where they rated statements about their experience with APPINITE on a 7-point Likert scale. The experimenter also had a short informal interview with each participant to solicit their comments and feedback.

4.4.4 Results

Overall, our participants had a good task completion rate. Among all 120 scenario instances across the 6 participants, 106 (87%) were successful in producing the intended target data description query on the first try. Note that we did not count retries caused by speech recognition errors, as it was not a focus of this study. Failed scenarios were all caused by incorrect or failed parsing of natural language instructions, which can be fixed by having a more robust natural language understanding mechanism. Participants successfully completed all initially failed scenarios in retries by rewording their verbal instructions after being prompted by APPINITE. Among all the 120 scenario instances, 24 instances required participants to have multi-turn conversations for disambiguation. 22 of these 24 (92%) were successful on the first try, and the rest were fixed by rewording.

In a survey on a 7-point Likert scale from "strongly disagree" to "strongly agree", our 6 participants found APPINITE "helpful in programming by demonstration" (mean = 7), "allowed them to express their intentions naturally" (mean = 6.8, $\sigma = 0.4$), and "easy to use" (mean = 7). They also agreed that "the multi-modal interface of APPINITE is helpful" (mean = 6.8, $\sigma = 0.4$), "the real-time visualization is helpful for disambiguation" (mean = 6.7, $\sigma = 0.5$), and "the error messages are helpful" (mean = 6.8, $\sigma = 0.4$).

4.4.5 Discussion

The study results suggested that APPINITE has good usability, and also that it has adequate performance for generating correct formal executable data description queries from demonstrations and natural language instructions in the sample scenarios.

Participants praised APPINITE's usefulness and ease of use. A participant reported that he found sample tasks very useful to have done by an intelligent agent. Participants also noted that without APPINITE, it would be almost impossible for end users without programming expertise to create automation scripts for these tasks, and it would also take considerable effort for experienced programmers to do so.

Our results illustrate the effectiveness of combining two input modalities, each with its own different of ambiguities, to more accurately infer user's intentions in EUD. A major challenge in EUD is that end users are unable to precisely specify their intended behaviors in formal language. Thus, easier-to-use but ambiguous alternative programming techniques like PBD and natural language programming are adopted. Our results suggest that end users can effectively clarify their intentions in a complementary technique with adequate guidance from the system when the initial input was ambiguous. Further research is needed on how users naturally select modalities in multi-modal environments, and on how interfaces can support more fluid transition between modalities.

Another insight from APPINITE is to leverage app GUIs as a shared grounding for EUD. By asking users to describe intentions in natural language referring to GUI contents, APPINITE's approach constrains the scope of instructions to a limited space, making semantic parsing feasible. Since users are already familiar with app GUIs, they do not have to learn new symbols or mechanisms as in scripting or visual languages. The knowledge graph extraction further provides users with greater expressive power by abstracting higher-level semantics from platform-specific implementations, enabling users to talk about semantic relations for the items such as "the cheapest restaurant" and "the score for Minnesota."

4.5 Chapter Conclusion

Natural language is a natural and expressive medium for end users to specify their intentions and can provide useful complementary information about user intentions when used in conjunction with other end user development approaches, such as programming by demonstration. This chapter described the new multi-modal APPINITE interface. It combines natural language instructions with demonstrations to provide end users with greater expressive power to create more generalized GUI automation scripts in SUGILITE, while retaining the usability and transparency.

Chapter 5

Learning Task Conditionals and Concepts

5.1 Introduction

Although the core SUGILITE system was effective in supporting end user development for task automation (as discussed in Section 3.5), a major challenge remains — the system needs to help non-programmers to specify conditional structures in programs. Many common tasks involve conditional structures, yet they are difficult for non-programmers to correctly specify using existing EUD techniques due to the great distance between how end users think about the conditional structures, and how they are represented in programming languages [224].

Another problem in programming conditionals is supporting the instruction of *concepts*. In our study (details in Section 5.2), we found that end users often refer to ambiguous or vague concepts (e.g., **cold** weather, **heavy** traffic) when naturally instructing conditionals in a task. Moreover, even if a concept may seem clear to a human, an agent may still not understand it due to the limitations in its natural language understanding techniques and pre-defined ontology.

This chapter will describe our work on enabling end users to augment task automation scripts with conditional structures and new concepts through a combination of natural language programming and programming by demonstration (PBD). We took a *user-centered design* approach, first studying how end users naturally describe tasks with conditionals in natural language in the context of mobile apps, and what types of tasks they are interested in automating.

This chapter is modified from the conference paper: Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 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)*.

Based on insights from the formative study (Section 5.2), we designed and implemented a new multi-modal conversational framework for SUGILITE named PUMICE[1] that allows end users to program tasks with flexible conditional structures and new concepts across diverse domains through spoken natural language instructions and demonstrations.

PUMICE extends the SUGILITE [159] system. A key novel aspect of PUMICE's design is that it allows users to first describe the desired program behaviors and conditional structures naturally in natural language at a high level, and then collaborate with an intelligent agent through multi-turn conversations to explain and to define any ambiguities, concepts, and procedures in the initial description as needed in a top-down fashion. Users can explain new concepts by referring to either previously defined concepts, or to the contents of the GUIs of third-party mobile apps. Users can also define new procedures by demonstrating using third-party apps, as described in previous sections. This approach facilitates effective program reuse in automation authoring, and provides support for a wide range of application domains, which are two major challenges in prior EUD systems. The results from the motivating study (Section 5.2) suggest that this paradigm is not only feasible but also natural for end users, which was supported by our lab usability study (Section 5.4.1).

This work builds upon recent advances in natural language processing to allow PUMICE's semantic parser to learn from users' flexible verbal expressions when describing desired program behaviors. Through PUMICE's mixed-initiative conversations with users, an underlying persistent knowledge graph is dynamically updated with new procedural (i.e., actions) and declarative (i.e., concepts and facts) knowledge introduced by users, allowing the semantic parser to improve its understanding of user utterances over time. This structure also allows for effective reuse of user-defined procedures and concepts at a fine granularity, reducing user effort in EUD.

PUMICE presents a multi-modal interface, through which users interact with the system using a combination of demonstrations, pointing, and spoken commands. Users may use any modality that they choose, so they can leverage their prior experience to minimize necessary training [188]. This interface also provides users with guidance through a mix of visual aids and verbal directions through various stages in the process to help users overcome common challenges and pitfalls identified in the formative study, such as the omission of else statements, the difficulty in finding correct GUI objects for defining new concepts, and the confusion in specifying proper data descriptions for target GUI objects. A summative lab usability study (Section 5.4.1) with 10 participants showed that users with little or no prior programming expertise could use PUMICE to program automation

[1] PUMICE is a type of volcanic rock. It is also an acronym for **P**rogramming in a **U**ser-friendly **M**ultimodal **I**nterface through **C**onversations and **E**xamples

scripts for 4 tasks derived from real-world scenarios. Participants also found PUMICE easy and natural to use.

5.2 Formative Study

As with the other parts of the system, we took a *user-centered* approach [210] for designing a natural end-user development system [211]. We first studied how end users naturally communicate tasks with declarative concepts and control structures in natural language for various tasks in the mobile app context through a formative study on Amazon Mechanical Turk with 58 participants (41 of whom were non-programmers; 38 identified as male, 19 identified as female, 1 identified as non-binary).

Each participant was presented with a graphical description of an everyday task for a conversational agent to complete in the context of mobile apps. All tasks had distinct conditions for a given task so that each task should be performed differently under different conditions, such as playing different genres of music based on the time of the day. Each participant was assigned to one of 9 tasks. To avoid biasing the language used in the responses, we used the Natural Programming Elicitation method [210] by showing graphical representations of the tasks with limited text in the prompts. Participants were asked how they would verbally instruct the agent to perform the tasks so that the system could understand the differences among the conditions and what to do in each condition. Each participant was first trained using an example scenario and the corresponding example verbal instructions.

To study whether having mobile app GUIs can affect users' verbal instructions, we randomly assigned participants into one of two groups. For the experimental group, participants instructed agents to perform the tasks while looking at relevant app GUIs. Each participant was presented with a mobile app screenshot with arrows pointing to the screen component that contained the information pertinent to the task condition. Participants in the control group were not shown app GUIs. At the end of each study session, we also asked the participants to come up with another task scenario of their own where an agent should perform differently in different conditions.

The participants' responses were analyzed by two independent coders using open coding [260]. The inter-rater agreement [61] was $\kappa = 0.87$, suggesting good agreement. 19% of responses were excluded from the analysis for quality control due to the lack of effort in the responses, question misunderstandings, and blank responses.

Here are the most relevant findings which motivated the design of PUMICE.

5.2.1 App GUI Grounding Reduces Unclear Concept Usage

We analyzed whether each user's verbal instruction for the task provided a clear definition of the conditions in the task. In the control group (instructing without seeing app screenshots), 33% of the participants used ambiguous, unclear or vague concepts in the instructions, such as "*If it is daytime, play upbeat music...*" which is ambiguous as to when the user considers it to be "daytime." This is despite the fact that the example instructions they saw had clearly defined conditions.

Interestingly, for the experimental group, where each participant was provided an app screenshot displaying specific information relevant to the task's condition, fewer participants (9%) used ambiguous or vague concepts (this difference is statistically significant with $p < 0.05$), while the rest clearly defined the condition (e.g., "*If the current time is before 7 pm...*"). These results suggest that end users naturally use ambiguous and vague concepts when verbally instructing task logic, but showing users relevant mobile app GUIs with concrete instances of the values can help them ground the concepts, leading to fewer ambiguities and vagueness in their descriptions. The implication is that a potentially effective approach to avoiding unclear utterances for agents is to guide users to explain them in the context of app GUIs.

5.2.2 Unmet User Expectation of Common Sense Reasoning

We observed that participants often expected and assumed the agent to have the capability of understanding and reasoning with common sense knowledge when instructing tasks. For example, one user said, "*if the day is a weekend*". The agent would therefore need to understand the concept of "weekend" (i.e., how to know today's day of the week, and what days count as "weekend") to resolve this condition. Similarly when a user talked about "sunset time", he expected the agent to know what it meant, and how to find out its value.

However, the capability for common sense knowledge and reasoning is very limited in current agents, especially across many diverse domains, due to the spotty coverage and unreliable inference of existing common sense knowledge systems. Managing user expectations and communicating the agent's capability is also a long-standing unsolved challenge in building interactive intelligent systems [177]. A feasible workaround is to enable the agent to ask users to ground new concepts to existing contents in apps when they come up, and to build up knowledge of concepts over time through its interaction with users.

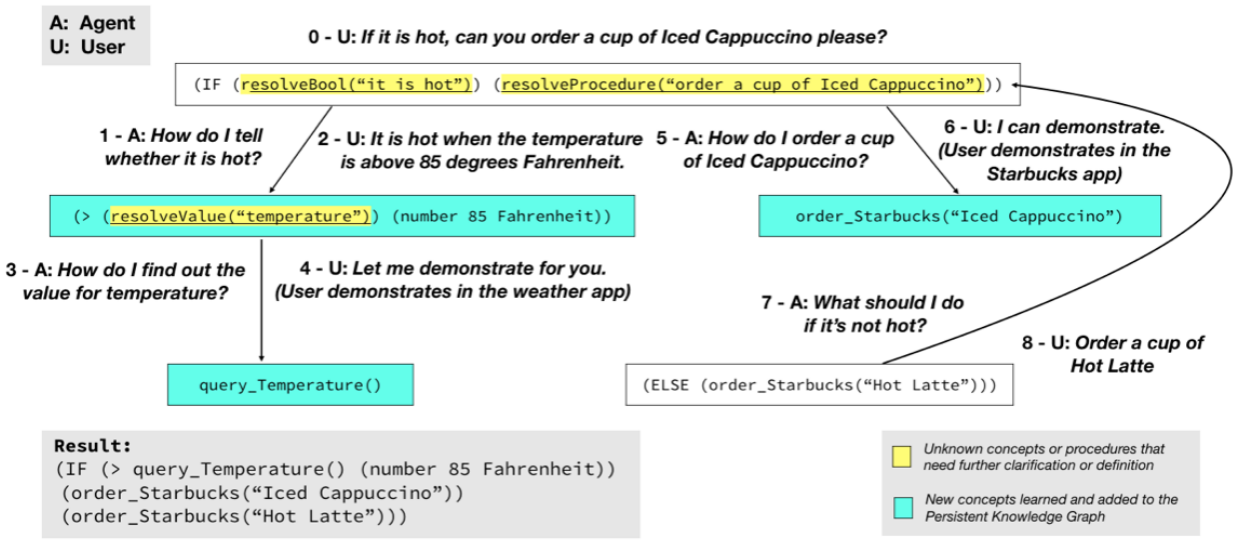


Figure 5.1: Example structure of how PUMICE learns the concepts and procedures in the command "If it's hot, order a cup of Iced Cappuccino." The numbers indicate the order of utterances. The screenshot on the right shows the conversational interface of PUMICE. In this interactive parsing process, the agent learns how to query the current temperature, how to order any kind of drink from Starbucks, and the generalized concept of "hot" as "a temperature (of something) is greater than another temperature".

5.2.3 Frequent Omission of Else Statements

In the study, despite all provided example responses containing else statements, 18% of the 39 descriptions from users omitted an else statement when it was expected. "*Play upbeat music until 8pm every day,*" for instance, may imply that the user desires an alternative genre of music to be played at other times. Furthermore, 33% omitted an else statement when a person would be expected to infer an else statement, such as: "*If a public transportation access point is more than half a mile away, then request an Uber*," which implies using public transportation otherwise. This might be a result of the user's expectation of common sense reasoning capabilities. The user omits what they expect the agent can infer to avoid prolixity, similar to patterns in human-human conversations [101].

These findings suggest that end users will often omit appropriate else statements in their natural language instructions for conditionals. Therefore, the agent should proactively ask users about alternative situations in conditionals when appropriate.

Motivated by the formative study results, we designed the PUMICE interface that supports understanding ambiguous natural language instructions for task automation by allowing users to recursively define any new, ambiguous or vague concepts in a multi-level top-down process.

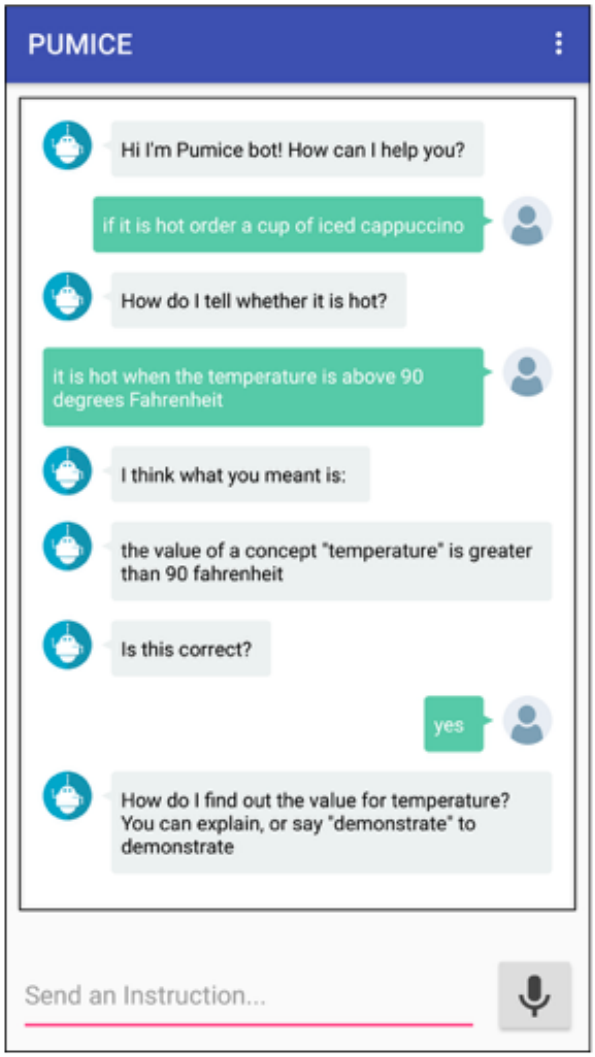


Figure 5.2: The conversational interface of PUMICE for the concept learning process shown in Figure 5.1.

5.3 PUMICE's Design

5.3.1 Example Usage Scenario

This section shows an example scenario to illustrate how PUMICE works[2]. Suppose a user starts teaching SUGILITE agent a new task automation rule through PUMICE interface by saying, "*If it's hot, order a cup of Iced Cappuccino.*" We also assume that the agent has no prior knowledge about the relevant task domains (weather and coffee ordering). Due to the lack of domain knowledge, the agent does not understand "it's hot" and "order a cup of Iced Cappuccino". However, the agent can recognize the conditional structure in the utterance (the parse for Utterance 0 in Figure 5.1) and can identify that "it's hot" should represent a Boolean expression while "order a cup of Iced Cappuccino" represents the action to perform if the condition is true.

PUMICE's semantic parser can mark unknown parts in the user-provided utterances using typed `resolve...()` functions, as marked in the yellow highlights in the parse for Utterance 0 in Figure 5.1. It then proceeds to ask the user to further explain these concepts. It asks, "*How do I tell whether it's hot?*" since it has already figured out that "it's hot" should be a function that returns a Boolean value. The user answers "*It is hot when the temperature is above 85 degrees Fahrenheit.*", as shown in Utterance 2 in Figure 5.1. The system understands the comparison (as shown in the parse for Utterance 2 in Figure 5.1), but does not know the concept of "temperature", only knowing that it should be a numeric value comparable to 85 degrees Fahrenheit. Hence it asks, "*How do I find out the value for temperature?*", to which the user responds, "*Let me demonstrate for you.*"

Here the user can demonstrate the procedure of finding the current temperature by opening the weather app on the phone, and pointing at the current reading of the weather. To assist the user, PUMICE uses a visualization overlay to highlight any GUI objects on the screen that fit into the comparison (i.e., those that display a value comparable to 85 degrees Fahrenheit). The user can choose from these highlighted objects (see Figure 5.3 for an example). Through this demonstration, PUMICE learns a reusable procedure `query_Temperature()` for getting the current value for the new concept *temperature*, and stores it in a persistent knowledge graph so that it can be used in other tasks. PUMICE confirms with the user every time it learns a new concept or a new rule, so that the user is aware of the current state of the system, and can correct any errors.

For the next phase, PUMICE has already determined that "order a cup of Iced Cappuccino" should be an action triggered when the condition "it's hot" is true, but does not know how to perform this action (also known as intent fulfillment in chatbots [168]). To learn how to perform this action, it asks, "*How do I order a cup of Iced Cappuccino?*", to which the user responds, "*I*

[2]A demo video is available at `https://www.youtube.com/watch?v=BAC2ZuJGY4M`

can demonstrate." The user then proceeds to demonstrate the procedure of ordering a cup of Iced Cappuccino using the existing app for Starbucks (a coffee chain). From the user's demonstration, PUMICE can figure out that "Iced Cappuccino" is a task parameter, and can learn the generalized procedure `order_Starbucks()` for ordering any item displayed in the Starbucks app, as well as a list of available items to order in the Starbucks app by looking through the Starbucks app's menus, using the underlying SUGILITE framework (see Sections 3.4 and 4.3.3) for processing the task recording.

Finally, PUMICE asks the user about the else condition by saying, "*What should I do if it's not hot?*" Suppose the user says "*Order a cup of Hot Latte*," then the user will not need to demonstrate again, because PUMICE can recognize "Hot Latte" as an available parameter option for the previously learned `order_Starbucks()` procedure.

5.3.2 Design Features

In this section, we discuss several of PUMICE's key design features in its user interactions, and how they were motivated by the results of the formative study.

Support for Concept Learning

In the formative study, we identified two main challenges with regard to concept learning. First, users often naturally use intrinsically unclear or ambiguous concepts when instructing intelligent agents (e.g., "*register for easy courses*", where the definition of "easy" depends on the context and the user's preference). Second, users expect agents to understand common-sense concepts that the agents may not know. To address these challenges, we designed and implemented the support for concept learning in PUMICE. PUMICE can detect and learn three kinds of unknown components in user utterances: *procedures*, *Boolean concepts*, and *value concepts*. Because PUMICE's support for procedure learning is unchanged from the underlying SUGILITE mechanisms (see Sections 3.4 and 4.3.3), we focus in this section on discussing how PUMICE learns Boolean concepts and value concepts.

When encountering an unknown or unclear concept in the utterance parsing result, PUMICE first determines the concept type based on the context. If the concept is used as a condition (e.g., "if **it is hot**"), then it should be of Boolean type. Similarly, if a concept is used where a value is expected (e.g., "if the **current temperature** is above 70°F" or "set the AC to the **current temperature**"), then it will be marked as a value concept. Both kinds of concepts are represented as typed `resolve()` functions in the parsing result (shown in Figure 5.1), indicating that they need to be further resolved down the line. This process is flexible. For example, if the user clearly defines

a condition without introducing unknown or unclear concepts, then PUMICE will not need to ask follow-up questions for concept resolution. This process uses a *lazy-evaluation* strategy, where the system waits for all unknown or unclear concepts to be defined before evaluating the expressions.

PUMICE recursively executes each `resolve()` function in the parsing result in a depth-first fashion. After a concept is fully resolved (i.e., all concepts used for defining it have been resolved), it is added to a persistent knowledge graph (details in Secion 5.4.2), and a link to it replaces the `resolve()` function. From the user's perspective, when a `resolve()` function is executed, the agent asks a question to prompt the user to further explain the concept. When resolving a Boolean concept, PUMICE asks, "*How do I know whether [concept_name]?*" For resolving a value concept, PUMICE asks, "*How do I find out the value of [concept_name]?*"

To explain a new Boolean concept, the user may verbally refer to another Boolean concept (e.g., "traffic is heavy" means "commute takes a long time") or may describe a Boolean expression (e.g., "the commute time is longer than 30 minutes"). When describing the Boolean expression, the user can use flexible words (e.g., colder, further, more expensive) to describe the relation (i.e., greater than, less than, and equal to). As explained previously, if any new Boolean or value concepts are used in the explanation, PUMICE will recursively resolve them. The user can also use more than one unknown value concepts, such as "if the price of a Uber is greater than the price of a Lyft" (Uber and Lyft are both popular ridesharing apps).

Similar to Boolean concepts, the user can refer to another value concept when explaining a value concept. When a value concept is concrete and available in a mobile app, the user can also demonstrate how to query the value through app GUIs. The formative study has suggested that this multi-modal approach is effective and feasible for end users. After users indicate that they want to demonstrate, PUMICE switches to the home screen of the phone, and prompts the user to demonstrate how to find out the value of the concept.

To help the user with value concept demonstrations, PUMICE highlights possible items on the current app GUI if the type of the target concept can be inferred from the type of the constant value, or using the type of value concept to which it is being compared (see Figure 5.3). For example, in the aforementioned "commute time" example, PUMICE knows that "commute time" should be a duration, because it is comparable to the constant value "30 minutes". Once the user finds the target value in an app, they can long press on the target value to select it and indicate it as the target value. PUMICE uses an interaction proxy overlay [304] for recording, so that it can record *all* values visible on the screen, not limited to the selectable or clickable ones. PUMICE can extract these values from the GUI using the screen scraping mechanism. Once the target value is selected, PUMICE stores the procedure of navigating to the screen where the target value is displayed and

finding the target value on the screen into its persistent knowledge graph as a value query, so that this query can be used whenever the underlying value is needed. After the value concept demonstration, PUMICE switches back to the conversational interface and continues to resolve the next concept if needed.

Concept Generalization and Reuse

Once concepts are learned, another major challenge is to generalize them so that they can be reused correctly in different contexts and task domains. This is a key design goal of PUMICE. It should be able to learn concepts at a fine granularity, and reuse parts of existing concepts as much as possible to avoid asking users to make redundant demonstrations. In the previous chapters, we focused on generalizing procedures, specifically learning parameters (Section 3.3.2) and intents for underlying operations (Chapter 4). In PUMICE, we explored the generalization of Boolean concepts and value concepts.

When generalizing Boolean concepts, PUMICE assumes that the Boolean operation stays the same, but the arguments may differ. For example, for the concept "hot" in Figure 5.1, 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 Figure 5.1, 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).

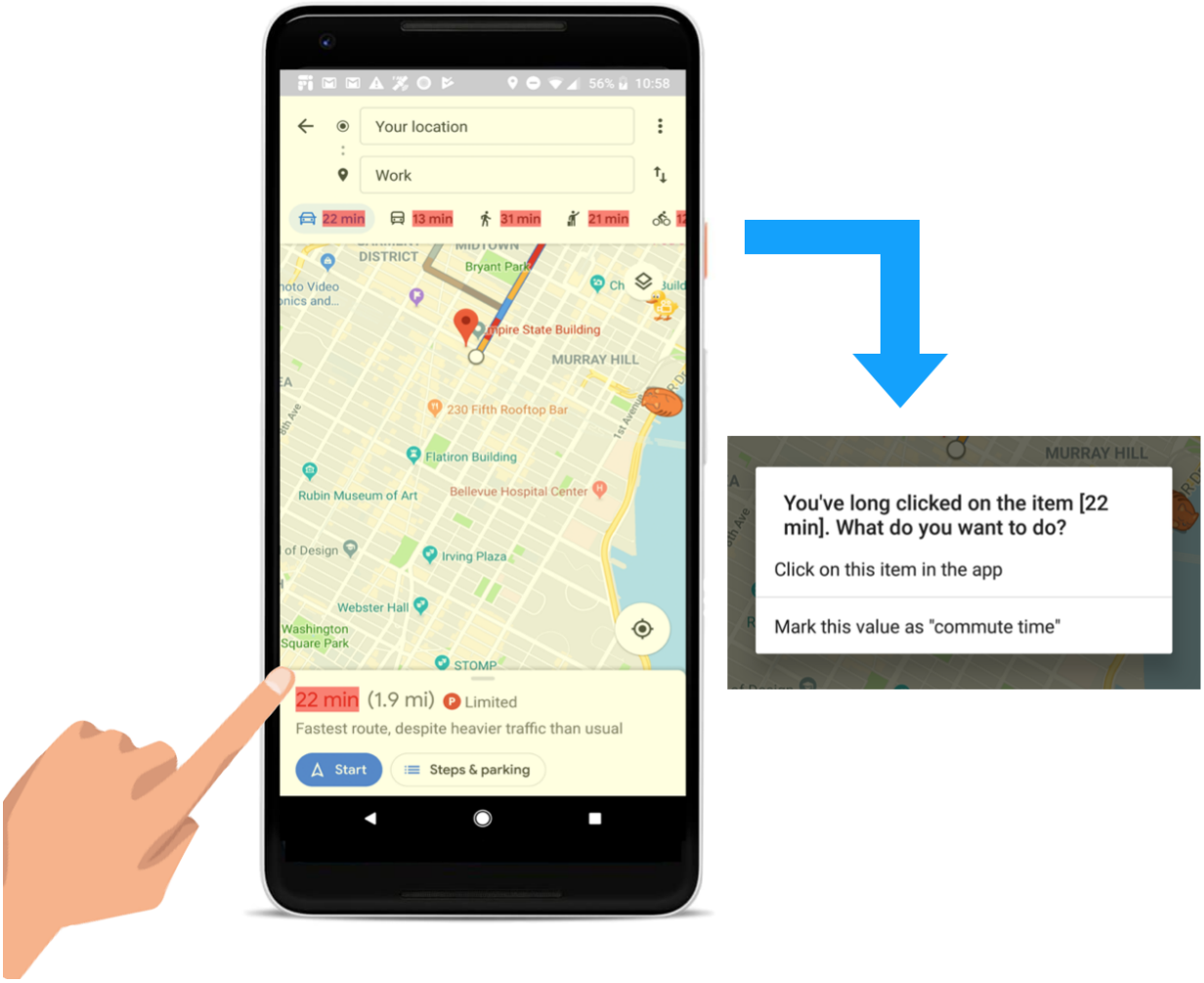


Figure 5.3: The user teaches the value concept "commute time" by demonstrating querying the value in Google Maps. The red overlays highlight all durations PUMICE was able to identify on the Google Maps GUI.

This mechanism allows 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).

Error Recovery and Backtracking

Like all other interactive EUD systems, it is crucial for PUMICE to properly handle errors, and to backtrack from errors in speech recognition, semantic parsing, generalizations, and inferences of intent. We iteratively tested early prototypes of PUMICE with users through early usability testing, and developed the following mechanisms to support error recovery and backtracking in PUMICE.

To mitigate semantic parsing errors, we implemented a mixed-initiative mechanism where PUMICE can ask users about *components* within the parsed expression if the parsing result is considered incorrect by the user. Because parsing result candidates are all typed expressions in PUMICE's internal functional domain-specific language (DSL) as a conditional, Boolean, value, or procedure, PUMICE can identify components in a parsing result that it is less confident about by comparing the top candidate with the alternatives and confidence scores, and ask the user about them.

For example, suppose the user defines a Boolean concept "good restaurant" with the utterance "the rating is better than 2". The parser is uncertain about the comparison operator in this Boolean expression, since "better" can mean either "greater than" or "less than" depending on the context. It will ask the user "*I understand you are trying to compare the value concept 'rating' and the value '2', should 'rating' be greater than, or less than '2'?*" The same technique can also be used to disambiguate other parts of the parsing results, such as the argument of `resolve()` functions (e.g., determining whether the unknown procedure should be "order a cup of Iced Cappuccino" or "order a cup" for Utterance 0 in Figure 5.1).

PUMICE also provides an "undo" function to allow the user to backtrack to a previous conversational state in case of incorrect speech recognition, incorrect generalization, or when the user wants to modify a previous input. Users can either say that they want to go back to the previous state, or click on an "undo" option in PUMICE's menu (this option can be activated from the option icon on the top right corner on the screen shown in Figure 5.2).

5.4 System Implementation

5.4.1 Semantic Parsing

We extended the semantic parser for PUMICE from the APPINITE parser (see Section 4.3.2) using the SEMPRE framework [24] to support the new lazy-evaluation top-down conversational framework for handling unknown concepts. The parser runs on a remote Linux server, and communicates with the PUMICE client through an HTTP RESTful interface. It uses the Floating Parser architecture, which is a grammar-based approach that provides more flexibility without requiring hand-engineering of lexicalized rules like synchronous CFG or CCG based semantic parsers [227]. This approach also provides more interpretable results and requires less training data than neural network approaches (e.g., [300, 301]). The parser parses user utterances into expressions in a simple functional DSL we created for PUMICE.

A key feature we added to PUMICE's parser is allowing typed `resolve()` functions in the parsing results to indicate unknown or unclear concepts and procedures. This feature adds interactivity to the traditional semantic parsing process. When this `resolve()` function is called at runtime, the front-end PUMICE agent asks the user to verbally explain, or to demonstrate how to fulfill this `resolve()` function. If an unknown concept or procedure is resolved through verbal explanation, the parser can parse the new explanation into an expression of its original type in the target DSL (e.g., an explanation for a Boolean concept is parsed into a Boolean expression), and replace the original `resolve()` function with the new expression. The parser also adds relevant utterances for existing concepts and procedures, and visible text labels from demonstrations on third-party app GUIs to its set of lexicons, so that it can understand user references to those existing knowledge and in-app contents. PUMICE's parser was trained on rich features that associate lexical and syntactic patterns (e.g., unigrams, bigrams, skipgrams, part-of-speech tags, named entity tags) of user utterances with semantics and structures of the target DSL over a small number of training data ($n = 905$) that were mostly collected and enriched from the formative study.

5.4.2 Knowledge Representations

PUMICE maintains two kinds of knowledge representations: a continuously refreshing UI snapshot graph representing third-party app contexts for demonstration, and a persistent knowledge base for storing learned procedures and concepts.

The purpose of the UI snapshot graph is to support understanding the user's references to app GUI contents in their verbal instructions. The UI snapshot graph mechanism used in PUMICE

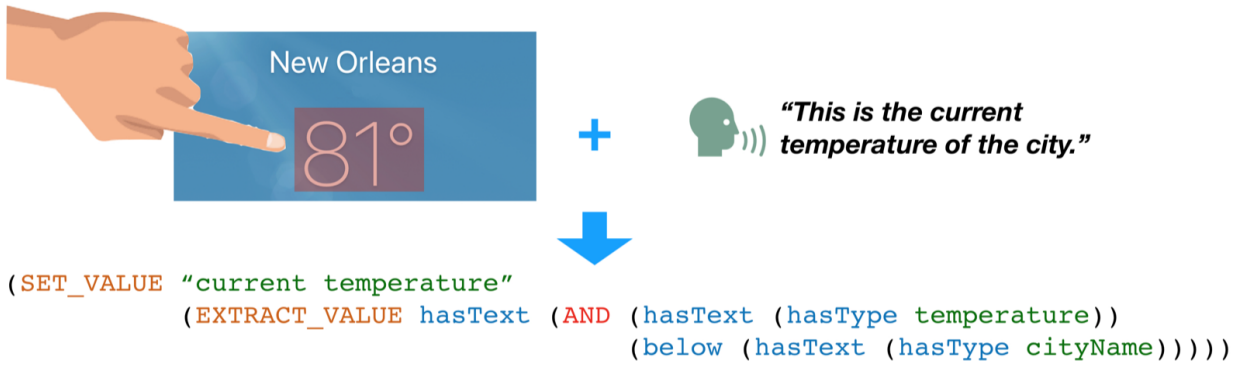


Figure 5.4: An example showing how PUMICE parses the user's demonstrated action and verbal reference to an app's GUI content into a SET_VALUE statement with a query over the UI snapshot graph when resolving a new value concept "current temperature."

was extended from APPINITE (see Section 4.3.1). For every state of an app's GUI, a UI snapshot graph is constructed to represent *all* visible and invisible GUI objects on the screen, including their types, positions, accessibility labels, text labels, various properties, and spatial relations among them. We used a lightweight semantic parser from the Stanford CoreNLP [191] to extract types of structured data (e.g., temperature, time, date, phone number) and named entities (e.g., city names, people's names). When handling the user's references to app GUI contents, PUMICE parses the original utterances into queries over the current UI snapshot graph (example in Figure 5.4). This approach allows PUMICE to generate flexible queries for value concepts and procedures that accurately reflect user intents, and which can be reliably executed in future contexts.

The persistent knowledge base is an add-on to the original SUGILITE system. It stores all procedures, concepts, and facts PUMICE has learned from the user. Procedures are stored as SUGILITE scripts, with the corresponding trigger utterances, parameters, and possible alternatives for each parameter. Each Boolean concept is represented as a set of trigger utterances, Boolean expressions with references to the value concepts or constants involved, and contexts (i.e., the apps and actions used) for each Boolean expression. Similarly, the structure for each stored value concept includes its triggering utterances, demonstrated value queries for extracting target values from app GUIs, and contexts for each value query.

5.5 User Study

We conducted a lab study to evaluate the usability of PUMICE. In each session, a user completed 4 tasks. For each task, the user instructed PUMICE to create a new task automation, with the required

conditionals and new concepts. We used a task-based method to specifically test the usability of PUMICE's design, since the motivation for the design derives from the formative study results. We did not use a control condition, as we could not find other tools that can feasibly support users with little programming expertise to complete the target tasks.

5.5.1 Participants

We recruited 10 participants (5 identified as female, 5 identified as male, ages 19 to 35) for this study. Each study session lasted 40 to 60 minutes. We compensated each participant $15 for their time. 6 participants were students in two local universities, and the other 4 worked different technical, administrative, or managerial jobs. All participants were experienced smartphone users who had been using smartphones for at least 3 years. 8 out of 10 participants had some prior experience of interacting with conversational agents like Siri, Alexa and Google Assistant.

We asked the participants to report their programming experience on a five-point scale from "never programmed" to "experienced programmer". Among our participants, there were 1 who had never programmed, 5 who had only used end-user programming tools (e.g., Excel functions, Office macros), 1 novice programmer with experience equivalent to 1-2 college-level programming classes, 1 programmer with 1-2 years of experience, and 2 programmers with more than 3 years of experience. In our analysis, we will label the first two groups "non-programmers" and the last three groups "programmers".

5.5.2 Procedure

At the beginning of each session, the participant received a 5-minute tutorial on how to use PUMICE. In the tutorial, the experimenter demonstrated an example of teaching PUMICE to check the bus schedule when "it is late", and "late" was defined as "current time is after 8pm" through a conversation with PUMICE. The experimenter then showed how to demonstrate to PUMICE finding out the current time using the Clock app.

Following the tutorial, the participant was provided a Google Pixel 2 phone with PUMICE and relevant third-party apps installed. The experimenter showed the participant the available apps, and made sure that the participant understood the functionality of each third-party app. We did this because the underlying assumption of the study (and the design of PUMICE) is that users are familiar with the third-party apps, so we are testing whether they can successfully use PUMICE, not the apps. Then, the participant received 4 tasks in random order. We asked participants to keep trying until they were able to correctly execute the automation, and that they were happy with

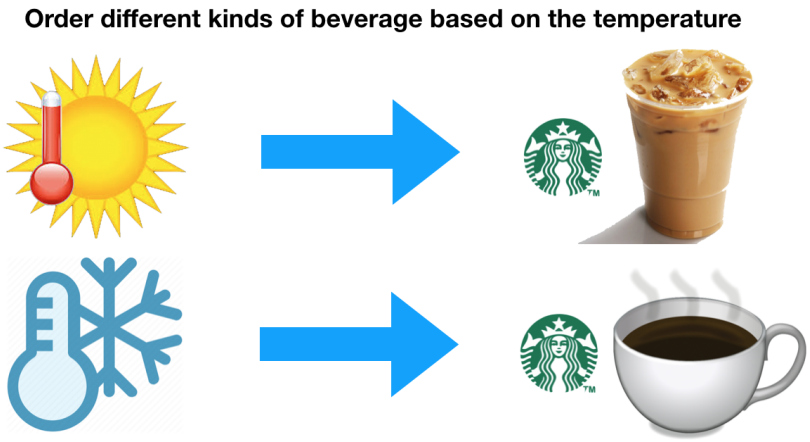


Figure 5.5: The graphical prompt used for Task 1 – A possible user command can be "*Order Iced coffee when it's hot outside, otherwise order hot coffee when the weather is cold.*"

the resulting actions of the agent. We also checked the scripts at the end of each study session to evaluate their correctness.

After completing the tasks, the participant filled out a post-survey to report the perceived usefulness, ease of use, and naturalness of interactions with PUMICE. We ended each session with a short informal interview with the participant on their experiences with PUMICE.

5.5.3 Tasks

We assigned 4 tasks to each participant. These tasks were designed by combining common themes observed in users' proposed scenarios from the formative study. We ensured that these tasks *(i)* covered key PUMICE features (i.e., concept learning, value query demonstration, procedure demonstration, concept generalization, procedure generalization and "else" condition handling); *(ii)* involved only app interfaces that most users are familiar with; and *(iii)* used conditions that we can control so we can test the correctness of the scripts (we controlled the temperature, the traffic conditions, and the room price by manipulating the GPS location of the phone).

In order to minimize biasing users' utterances, we used the Natural Programming Elicitation method [210]. Task descriptions were provided in the form of graphics, with minimal text descriptions that could not be directly used in user instructions (see Figure 5.5 for an example).

Task 1 In this task, the user instructs PUMICE to order iced coffee when the weather is hot, and order hot coffee otherwise (Figure 5.5). We pre-taught PUMICE the concept of "hot" in the task domain of turning on the air conditioner. So the user needs to utilize the concept generalization

feature to generalize the existing concept "hot" to the new domain of coffee ordering. The user also needs to demonstrate ordering iced coffee using the Starbucks app, and to provide "order hot coffee" as the alternative for the "else" operation. The user does not need to demonstrate again for ordering hot coffee, as it can be automatically generalized from the previous demonstration of ordering iced coffee.

Task 2 In this task, the user instructs PUMICE to set an alarm for 7am if the traffic is heavy on their commuting route. We pre-stored "home" and "work" locations in Google Maps. The user needs to define "heavy traffic" as prompted by PUMICE by demonstrating how to find out the estimated commute time, and explaining that "heavy traffic" means that the commute takes more than 30 minutes. The user then needs to demonstrate setting a 7am alarm using the built-in Clock app.

Task 3 In this task, the user instructs PUMICE to choose between making a hotel reservation and requesting a Uber to go home depending on whether the hotel price is cheap. The user should verbally define "cheap" as "room price is below $100", and demonstrate how to find out the hotel price using the Marriott (a hotel chain) app. The user also needs to demonstrate making the hotel reservation using the Marriott app, specify "request an Uber" as the action for the "else" condition, and demonstrate how to request an Uber using the Uber app.

Task 4 In this task, the user instructs PUMICE to order a pepperoni pizza if there is enough money left in the food budget. The user needs to define the concept of "enough budget", demonstrate finding out the balance of the budget using the Spending Tracker app, and demonstrate ordering a pepperoni pizza using the Papa Johns (a pizza chain) app.

5.5.4 Results

All participants were able to complete all 4 tasks. The total time for tasks ranged from 19.4 minutes to 25 minutes for the 10 participants. Figure 5.6 shows the overall average task completion time of each task, as well as the comparison between the non-programmers and the programmers. The average total time-on-task for programmers (22.12 minutes, SD=2.40) was slightly shorter than that for non-programmers (23.06 minutes, SD=1.57), but the difference was not statistically significant.

Most of the issues encountered by participants were actually from the Google Cloud speech recognition system used in PUMICE. It would sometimes misrecognize the user's voice input, or

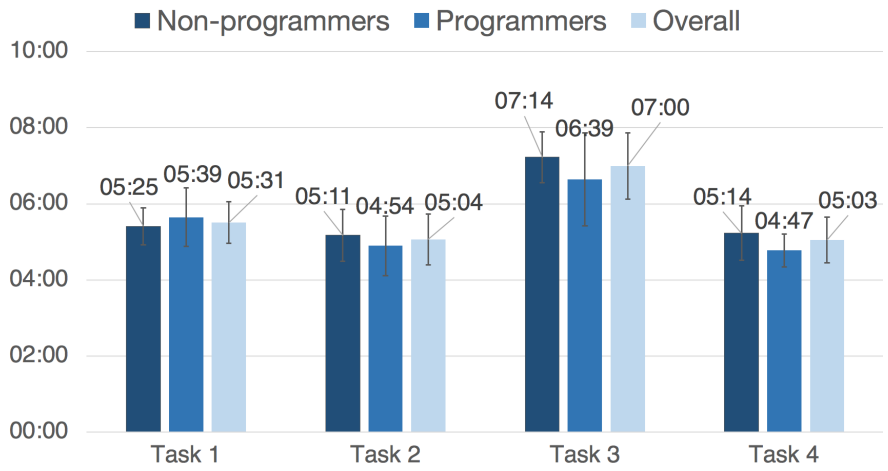


Figure 5.6: The average task completion times for each task. The error bars show one standard deviation in each direction.

cut off the user early. These errors were handled by the participants using the "undo" feature in PUMICE. Some participants also had parsing errors. PUMICE's current semantic parser has limited capabilities in understanding references of pronouns (e.g., for an utterance *"it takes longer than 30 minutes to get to work"*, the parser would recognize it as "it" instead of "the time it takes to get to work" is greater than 30 minutes). Those errors were also handled by participants through undoing and rephrasing. One participant ran into the "confusion of Boolean operator" problem in Task 2 when she used the phrase "*commute [time is] worse than 30 minutes*", for which the parser initially recognized incorrectly as "commute is *less than* 30 minutes." She was able to correct this error by specifying that she meant "greater than" for the word "worse" using the error recovery mechanism discussed in Section 5.3.2.

Overall, no participant had major problems with the multi-modal interaction approach and the top-down recursive concept resolution conversational structure, which was encouraging. However, all participants had received a tutorial with an example task demonstrated. We also emphasized in the tutorial that they should try to use concepts that can be found in mobile apps in their explanations of new concepts. These factors might contributed to the success of our participants.

In a post survey, we asked our participants to rate statements about the usability, naturalness, and usefulness of PUMICE on a 7-point Likert scale ranging from "strongly disagree" to "strongly agree". PUMICE scored on average 6.2 on "*I feel* PUMICE *is easy to use*", 6.1 on "*I find my interactions with* PUMICE *natural*", and 6.9 on "*I think* PUMICE *is a useful tool for automating tasks on smartphones,*". The results indicated that our participants were generally satisfied with their experience using PUMICE.

5.5.5 Discussion

In the informal interview after completing the tasks, participants praised PUMICE for its naturalness and low learning barriers. Non-programmers were particularly impressed by the multi-modal interface. For example, P7 (who was a non-programmer) said: "*Teaching* PUMICE *feels similar to explaining tasks to another person...[Pumice's] support for demonstration is very easy to use since I'm already familiar with how to use those apps.*" Participants also considered PUMICE's top-down interactive concept resolution approach very useful, as it does not require them to define everything clearly upfront.

Participants were excited about the usefulness of PUMICE. P6 said, "*I have an Alexa assistant at home, but I only use them for tasks like playing music and setting alarms. I tried some more complicated commands before, but they all failed. If it had the capability of* PUMICE*, I would definitely use it to teach Alexa more tasks.*" They also proposed many usage scenarios based on their own needs in the interview, such as paying off credit card balance early when it has reached a certain amount, automatically closing background apps when the available phone memory is low, monitoring promotions for gifts saved in the wish list when approaching anniversaries, and setting reminders for events in mobile games.

Several concerns were also raised by our participants. P4 commented that PUMICE should "just know" how to find out weather conditions without requiring her to teach it since "all other bots know how to do it", indicating the need for a hybrid approach that combines EUD with pre-programmed common functionalities. P5 said that teaching the agent could be too time-consuming unless the task was very repetitive since he could just "do it with 5 taps." Several users also expressed privacy concerns after learning that PUMICE can see all screen contents during demonstrations, while one user, on the other hand, suggested having PUMICE observe him at all times so that it can learn things in the background.

5.6 Chapter Conclusion

This chapter presented PUMICE, a new multi-modal conversational framework for SUGILITE that allows it to learn task conditionals and relevant concepts from conversational natural language instructions and demonstrations. The design of PUMICE showcased the idea of using multi-modal interactions to support the learning of unknown, ambiguous, or vague concepts in users' verbal commands, which was shown to be common in the formative study.

In PUMICE's approach, users can explain abstract concepts in task conditions using more concrete smaller concepts, and ground them by demonstrating with third-party mobile apps. More

broadly, this work demonstrates how combining conversational interfaces and demonstrational interfaces can create easy-to-use and natural end user development experiences.

Chapter 6

Applications in Smart Home Scenarios

6.1 Introduction

In recent years, the rapid growth of Internet of Things (IoT) has surrounded users with various smart appliances, sensors, and devices. Through their connections, these smart objects can understand and react to their environment, enabling novel computing applications [239]. A past study has shown that users have highly diverse and personalized desired behaviors for their smart home automation, and, as a result, they need end-user tools to enable them to program their environment [273]. Especially with the growing number of devices, the complexity of the systems, and their importance in everyday life, it is increasingly important to enable end users to create the programs themselves for those devices to achieve the desired user experience [198, 244].

Many manufacturers of smart devices have provided their customers with tools for creating their own automation. For example, LG has the SmartThinQ app, where the user can author schedules and rules for their supported LG smart appliances such as fridges, washers, and ovens. Similar software is also provided by companies like Samsung (SmartThings), Home Depot (Wink), and WeMo. However, a major limitation for all of these is the lack of interoperability and compatibility with devices from other manufacturers. They all only support the limited set of devices manufactured by their own companies or their partners. Therefore, users are restricted to creating automation using devices within the same "ecosystem" and are unable to, for instance, create automation to adjust the setting of an LG air conditioner based on the reading from a Samsung sensor.

This chapter is modified from the conference paper: Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A. Myers. Programming IoT Devices by Demonstration Using Mobile Apps. In *Proceedings of the International Symposium on End User Development (IS-EUD 2017)*.

Some platforms partially address this problem. For example, IFTTT[1] is a popular service that enables end users to program in the form of "if `trigger`, then `action`" for hundreds of popular web services, apps, and IoT devices. With the help of IFTTT, the user can create automation across supported devices like a GE dishwasher, a WeMo coffeemaker, and a Fitbit fitness tracker. However, the applicability of IFTTT is still limited to devices and services offered by its partners, or those that provide open APIs which can connect to IFTTT. Even for the supported devices and services, often only a subset of the most commonly used functions is made available due to the required engineering effort. Other platforms like Apple HomeKit and Google Home also suffer from the same limitations. Because of the lack of a standard interface or a standard protocol, many existing tools and systems cannot support the heterogeneity of IoT devices [94, 103]. While generalized architectures for programming IoT devices and higher-level representations of IoT automation rules and scripts have been proposed (e.g., [84, 89, 103, 103, 231]), they have not yet been widely adopted in the most popular commercial IoT products, and there is some reason for pessimism that such an agreement will ever happen.

To address these problems, we have created an extension to the base version of SUGILITE named EPIDOSITE[2]. EPIDOSITE enables the creation of automation for IoT devices from different ecosystems by demonstration through manipulating their corresponding mobile apps using SUGILITE. Our system particularly targets the development of automation scripts for consumer IoT devices in the smart home environment by end users with little or no programming expertise. Thanks to the ubiquity of smartphones, for the majority of consumer IoT devices, there are corresponding smartphone apps available for remote controlling them, and these apps often have access to the full capabilities of the devices. A smartphone loaded with these apps is an ideal universal interface for monitoring and controlling the smart home environment [308]. Thus, by leveraging the smartphone as a hub, we can both read the status of the IoT sensors by scraping information from the user interfaces of their apps, and control the IoT actuators by manipulating their apps. To our knowledge, EPIDOSITE is the first end user development system for IoT devices using this approach.

6.2 EPIDOSITE's Advantages

EPIDOSITE's approach has the following three major advantages:

[1] https://ifttt.com/

[2] EPIDOSITE is a type of rock. The acronym stands for **E**nabling **P**rogramming of **I**oT **D**evices **O**n **S**martphone **I**nterfaces for **T**he **E**nd-users

- **Compatibility:** Unlike other EUD tools for consumer IoT devices, which can only support programming devices from the same company, within the same ecosystem, or which provide open access to their APIs, EPIDOSITE can support programming for most of the available consumer IoT devices if they provide Android apps for remote control. For selected phones with IR blasters (e.g., LG G5, Samsung Galaxy S6, HTC One) and the corresponding IR remote apps, EPIDOSITE can even control "non-smart" appliances such as TVs, DVRs, home theaters, and air conditioners that support IR remote control (but this obviously only works when the phone is aimed at the device).

- **Interoperability:** EPIDOSITE can support creating automation across multiple IoT devices, even if they are from different manufacturers. Besides IoT devices, EPIDOSITE can also support the incorporation of arbitrary third-party mobile apps and hundreds of external web services into the scripts. The exchange of information between devices is also supported by demonstration. The user can extract values of the readings and status of IoT devices using gestures on one mobile app, and use the values later as input for other apps. This approach addresses the challenge of supporting impromptu interoperability [76], a vision that devices acquired at different times, from different vendors and created under different design constraints and considerations should interconnect with no advance planning or implementation.

- **Usability:** We believe that EPIDOSITE should be easy to use, even for end users with little or no prior programming experience. Since the users are already familiar with how to use the mobile apps to monitor and to control IoT devices, EPIDOSITE minimizes the learning barrier by enabling users to use those familiar app interfaces to develop automation by demonstrating the procedures to control the IoT devices to perform the desired behaviors. A major advantage of PBD is that it can empower users while minimizing learning of programming concepts [65, 175]. The evaluation of the core SUGILITE system (see Section 3.5) has shown that most end users can successfully automate their tasks by demonstration using mobile apps on smartphones.

6.3 Example Usage Scenario

In this section, we use an example scenario to illustrate the procedure of creating an automation script with EPIDOSITE. For the scenario, we will create a script that turns on the TV set top box and turns off the light when someone enters the TV room. We will use a Verizon TV set-top box,

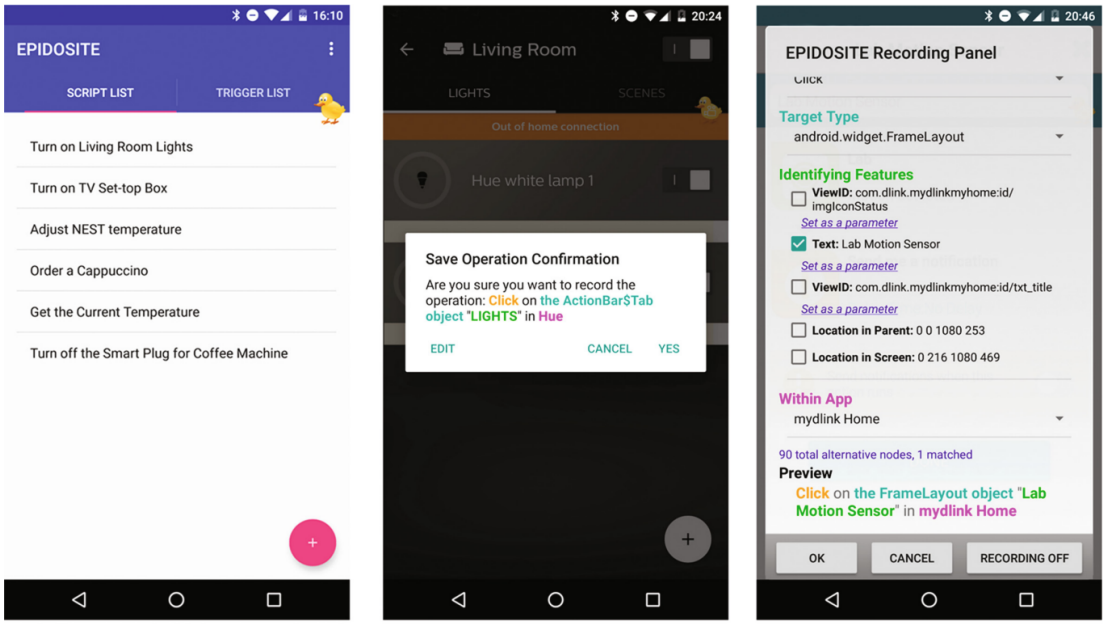


Figure 6.1: Screenshots of EPIDOSITE: (a) the main screen of EPIDOSITE showing a list of available scripts; (b) the confirmation dialog for an operation; (c) the data description editing panel for one operation. *Note that in the latest version, the data description editing panel (c) has been replaced with* APPINITE *(see Chapter 4).*

a Philips Hue Go light, and a D-Link DCH-S150 Wi-Fi motion sensor in this script. To our best knowledge, there exists no other EUD solution that can support all the above three devices.

First, the user starts EPIDOSITE (Figure 6.1a), creates a new script, and gives it a name. The phone then switches back to the home screen, and prompts the user to start demonstrating. The user now starts demonstrating how to turn off the light using the Philips Hue app — tapping on the Philips Hue icon on the home screen, choosing "Living Room", clicking on the "SCENES" tab, and selecting "OFF", which are the exact same steps as when the user turns off the light manually using the same app. After each action, the user can see a confirmation dialog from EPIDOSITE (Figure 6.1b). Running in the background as an Android accessibility service, EPIDOSITE can automatically detect the user's action and determine the features to use to identify the element using a set of heuristics, but the user can also manually edit the details of each recorded action in an editing panel (Figure 6.1c). Figure 6.2a shows the textual representation of the EPIDOSITE script created for turning off the light. Next, the user demonstrates turning on the TV set-top box using the SURE Universal Remote app on the phone (or other IR remote apps that support the Verizon TV set-top box), and EPIDOSITE records the procedure for that as well.

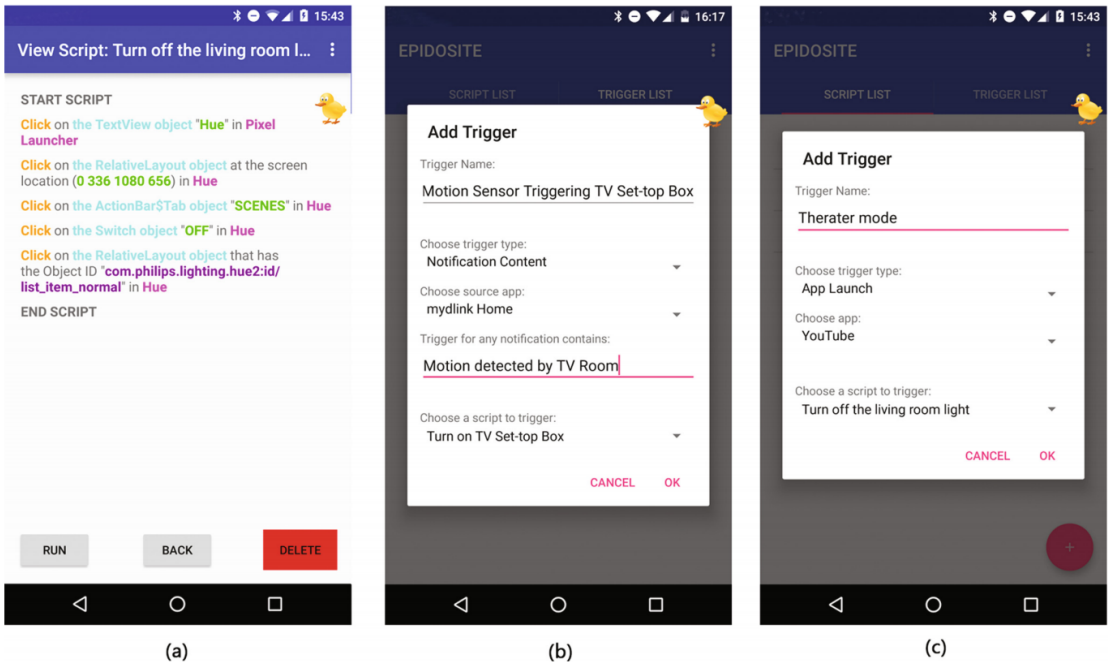


Figure 6.2: Screenshots of EPIDOSITE's script view and trigger creation interfaces: (a) the script view showing the script from the example usage scenario; (b) the window for creating an app notification trigger; (c) the window for creating an app launch trigger.

The user ends the demonstration recording and goes back to the EPIDOSITE app. She then clicks on the menu, and chooses "Add a Script Trigger". In the pop-up window (Figure 6.2b), she gives the trigger a name, selects "Notification" as the type of the trigger, specifies "mydlink Home" (the mobile app for the D-Link motion sensor) as the source app, chooses the script she just created as the script to trigger, enters "Motion detected by TV room" as the activation string, and finally, clicks on "OK" to save the trigger. This trigger will execute the script every time that the "mydlink Home" app sends a notification that contains the string "Motion detected by TV room".

The steps shown above are the whole process to create this automation. Once the trigger is enabled, the background Android accessibility service waits for the activation of the trigger. When the motion sensor detects a motion, an Android notification is generated and displayed by the mydlink Home app. Then EPIDOSITE intercepts this notification, activates the trigger, executes the script, and manipulates the UI of the Philips Hue app and the SURE Universal Remote app to turn off the lights and to turn on the TV set-top box.

6.4 EPIDOSITE's Key Features

6.4.1 Notification and App Launch Triggers

The most common context-aware applications in the smart home are naturally described using rule-based conditions in the model of trigger-action programming, where a *trigger* describes a condition, an event, or a scenario, and an *action* specifies the desired behavior when its associated trigger is activated [70, 273]. In EPIDOSITE, scripts can be triggered by the content of Android notifications, or as a result of the launch of a specified app on the phone.[3]

EPIDOSITE keeps a background Android accessibility service running at all times. Through the Android accessibility API, the service is notified about the system notifications and app launches. If the content of a new notification contains the activation string of a stored notification trigger (as shown in the example usage scenario using the interface shown in Figure 6.2), or an app associated with an app launch trigger has just launched, the corresponding automation script for the trigger will be executed. Figure 6.2c shows the interface with which the user can create an automation that turns off the light when the YouTube app launches, after first creating the "Turn off the living room light" script by demonstration.

These features allow scripts to be triggered not only by mobile apps for IoT devices, as shown in the example usage scenario, but also by other third-party Android apps. Prior research has shown that the usage of smartphone apps is highly contextual [42, 123] and also varies [307] for different groups of users. By allowing the launching of apps and the notifications from apps to trigger IoT scripts, the user can create highly context-aware automation with EPIDOSITE, for example, to change the color of the ambient lighting when the Music Player is launched, to adjust the thermostat when the Sleep Monitor app is launched, or even to warm up the car when the Alarm app rings (and sends the notification) on winter mornings.

Using the above two types of triggers, along with the external service trigger introduced in the next section, user-demonstrated scripts for smartphone apps can also be triggered by readings and status of IoT sensors and devices.

6.4.2 External Service Triggers

To expand the capabilities of EPIDOSITE to leverage all of the available resources, we implemented an optional server application that allows EPIDOSITE to integrate with the popular automation service IFTTT. Through this integration, an EPIDOSITE script can be triggered by over 360 web ser-

[3] The notification-based and app launch-based trigger mechanisms have not been integrated with the app interface-based conditional mechanism described in Chapter 5.

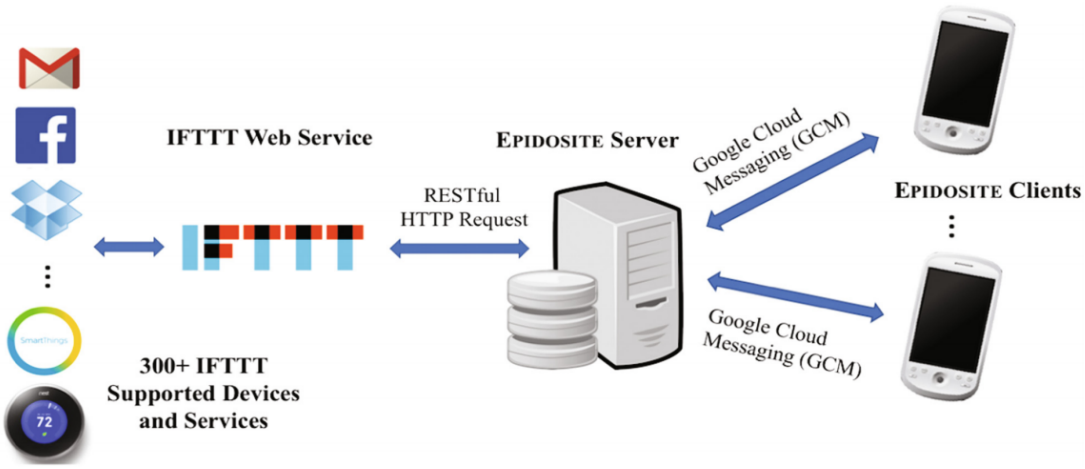


Figure 6.3: The architecture of EPIDOSITE's external service trigger mechanism.

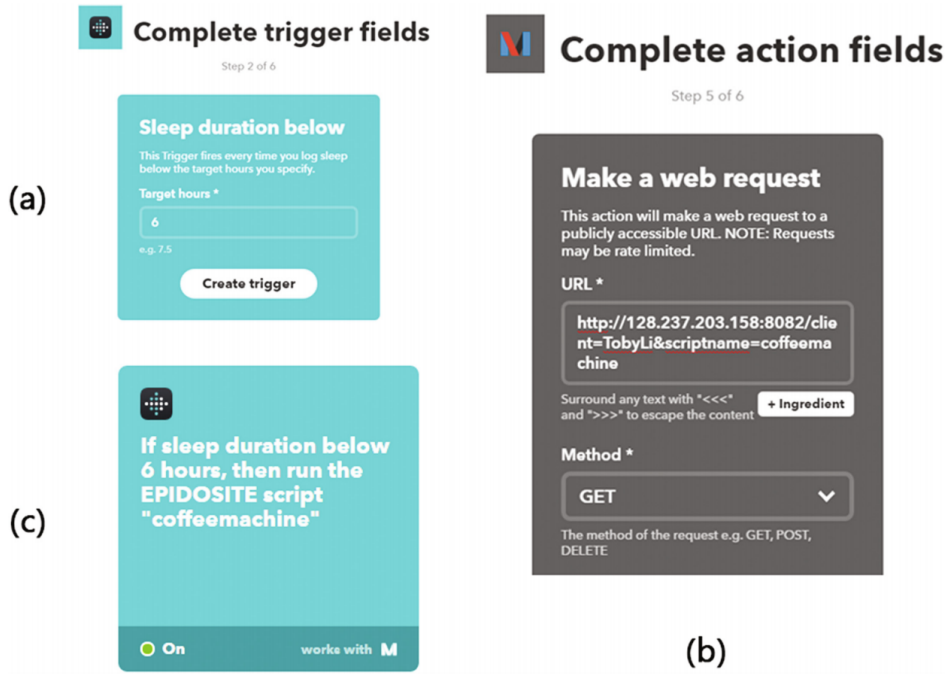


Figure 6.4: Creating an IFTTT applet that triggers an EPIDOSITE script: (a) creating the trigger condition "sleep duration below 6 hours" using the Fitbit activity tracker; (b) creating the action of running the EPIDOSITE script "coffeemachine" using the URL generated by EPIDOSITE; (c) the IFTTT applet created.

vices supported by IFTTT, including social networks (e.g., Facebook, Twitter), news (e.g., ESPN, Feedly), email, calendar management, weather and supported devices like smart hubs (e.g., Google Home, Amazon Echo), smart appliances, fitness trackers, home monitors, and smart speakers. An EPIDOSITE script can also be used to trigger actions for IFTTT-supported services. Figure 6.3 shows the overall architecture for supporting external service triggers, consisting of the client-side, the server-side, the IFTTT service, and how they communicate.

An IFTTT applet consists of two parts: a trigger and an action., in which either part can be an EPIDOSITE script. If an EPIDOSITE script is used as the trigger, then an HTTP request will be sent out to IFTTT via the EPIDOSITE server to execute the corresponding IFTTT action when the trigger is activated. Similarly, if an EPIDOSITE script is used as the action, then it will be executed on the corresponding client smartphone upon the client application receiving a Google Cloud Messaging (GCM) message sent by the EPIDOSITE server when the associated IFTTT trigger is activated. The EPIDOSITE server communicates with IFTTT through the IFTTT Maker channel, which can receive and make RESTful HTTP web requests. The EPIDOSITE server-side application is also highly scalable and can handle multiple clients at the same time.

To create an IFTTT triggered script, the user first creates an EPIDOSITE script for the "action" part by demonstration, where the user records the procedure of performing the desired task by manipulating the phone apps. Then, the user goes to IFTTT, chooses "New Applet," and chooses a trigger for the script. After this, the user chooses the Maker channel as the action. For the address for the web request, the EPIDOSITE app on the phone will automatically generate a URL which the user can just paste into this field. The auto-generated URL is in the format of:

```
http://[SERVER_ADDRESS]/client=[CLIENT_NAME]&scriptname=[SCRIPT_NAME]
```

where `[SERVER_ADDRESS]` is the address of the EPIDOSITE server, `[CLIENT_NAME]` is the name of the EPIDOSITE client (which by default is the combination of the phone owner's name and the phone model. e.g., "Amy's Nexus 6") and `[SCRIPT_NAME]` is the name of the EPIDOSITE script to trigger. The user can just paste this URL into the IFTTT field (see Figure 6.4).

The procedure to create an EPIDOSITE-triggered IFTTT applet is similar, except that the user needs to add "trigger an IFTTT applet" as an operation when demonstrating the EPIDOSITE script, and then use the Maker channel as the trigger.

6.4.3 Cross-app Interoperability

Interoperability among IoT devices has been a long-time important challenge [76]. Sharing data across devices from different "ecosystems" often requires the user to manually setup the connec-

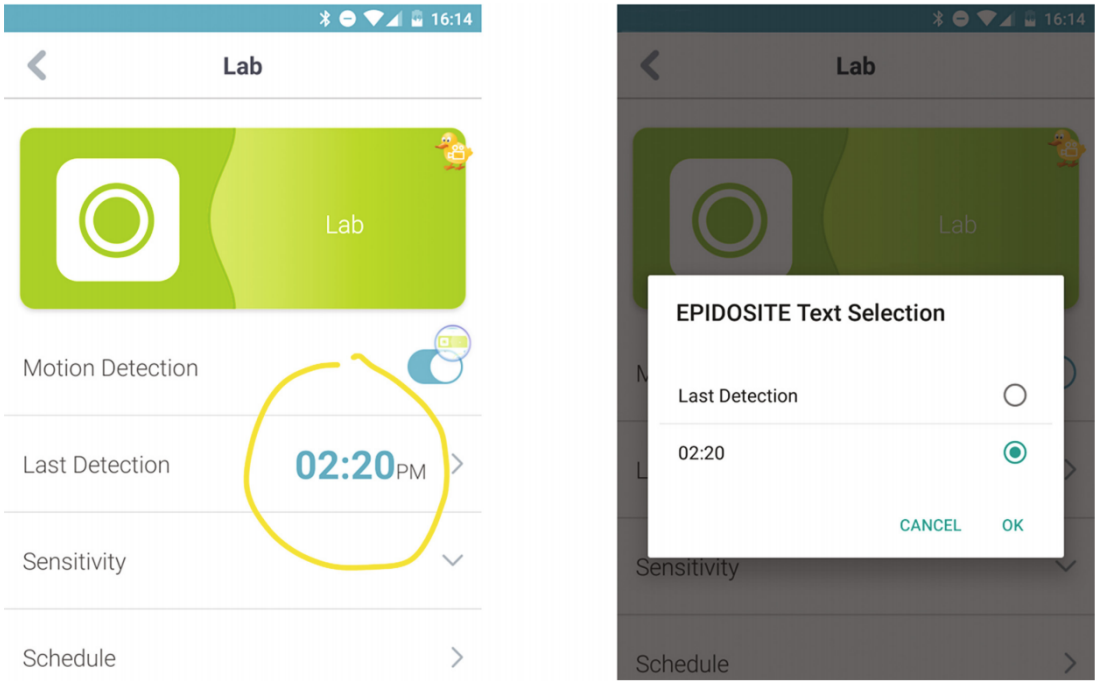


Figure 6.5: Extracting the time of the last detection from a D-link motion sensor in the Mydlink Home app using a circle gesture in EPIDOSITE.

tion using techniques like HTTP requests, which require careful planning and extensive technical expertise [68]. Middleware like [29, 89, 131, 255] supports IoT interoperation and provides a high-level representation model for common IoT devices, but these also require the user to have sophisticated programming skills.

EPIDOSITE supports the user in extracting the value of a `TextView` object in the user interface of an app by using a gesture during the demonstration, storing the value in a variable, and then using the values saved in these variables later in the script. All the user needs to do to save a value is to click on the "Get a Text Element on the Screen" option from the recording menu while demonstrating, circle the desired element on the screen using a finger gesture (the yellow stroke shown in Figure 6.5), and select the element in a pop-up menu (see Figure 6.5). Later, when the user needs to enter a string in a script, the user can instead choose to use the value from a previously created variable. Eventually, this value selection and extraction mechanism should be connected with the concept learning mechanism described in Section 5.3.2.

When a script that contains a value extraction operation is executed, EPIDOSITE will automatically navigate as demonstrated to the user interface where the desired value is located, and then dynamically extract the value using the appropriate data description query (see Chapter 4). This

approach does not require the involved app to have an API or other data export mechanism. As long as the desired value is displayed as a text string in the app, the value can be extracted and used in other parts of the script.

6.5 System Implementation and Technical Limitations

The client component of EPIDOSITE is an Android app extension to the base version of SUGILITE (see Chapter 3). The client component is standalone. There is also an optional server application available for supporting automation triggered by external web services through IFTTT. The server application is implemented in Java with Jersey[4] and Grizzly.[5]

On top of the base version of SUGILITE, EPIDOSITE added new features and mechanisms to support the programming of IoT devices in the smart home setting, including new ways for triggering scripts, new ways for scripts to trigger external services, and devices, and new mechanisms for sharing information among devices. To better meet the needs of developing for IoT devices, EPIDOSITE also supported programming for different devices in separated subscripts, and reusing the subscripts in new scripts. For example, the user can demonstrate the two scripts for "turning off the light" and "turning on the TV", and then create a new script of "if ..., turn off the light and turn on the TV") without having to demonstrate the procedures for performing the two tasks again.

The current version of EPIDOSITE has several technical limitations. First, for executing automation, the phone must be powered on and concurrently connected to all the devices involved in the automation. If the phone is powered off, or disconnected from the involved IoT devices, the automation will fail to execute. This limitation will particularly affect EPIDOSITE's applicability for devices that are connected to the phone via a local network or through a short-range wireless communication technology (e.g., Bluetooth, ZigBee, IR), since with these devices, the phone is restricted to be connected to the local network, or physically within range of the wireless connection for the automation to work. Second, EPIDOSITE automation needs to run in the foreground on the main UI thread of the phone. Thus, if automation is triggered when the user is actively operating the phone at the same time (e.g., if the user is on a phone call), then the user's interaction with the phone will be interrupted. The automation execution may also fail if it is interrupted by a phone event (e.g., an incoming phone call) or by the user's action.

For the above limitations, an approach is to use a separate phone as the hub for IoT automation, and to run EPIDOSITE on that phone instead of using the user's primary smartphone. By doing this,

[4]https://jersey.java.net/
[5]https://grizzly.java.net/

the separate phone can be consistently plugged in and stay connected with the IoT devices to ensure that the automation can be triggered and executed properly. Currently, a compatible Android phone can be purchased for less than $50, which makes this approach affordable.

6.6 Chapter Conclusion

In this chapter, we described EPIDOSITE, a SUGILITE extension that makes it possible for end users to create automation for consumer IoT devices on their smartphones. It supports programming across multiple IoT devices and exchanging information among them without requiring the devices to be of the same brand or within the same "ecosystem". The programming by demonstration approach minimizes the necessity to learn programming concepts. EPIDOSITE also supports using arbitrary third-party Android mobile apps and hundreds of available web services in the scripts to create highly context-aware and responsive automation.

Chapter 7

Breakdown Repairs in Task-Oriented Dialogues

7.1 Introduction

Conversational user interfaces have become increasingly popular and ubiquitous in our everyday lives, assisting users with tasks from diverse domains. However, despite the advances in their natural language understanding capabilities, prevailing conversational systems, including SUGILITE, are still far from being able to understand the wide range of flexible user utterances and engage in complex dialog flows [102]. 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 [19, 124]. 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 [22, 64, 121, 122, 185], and may cause users to abandon the current task [8]. In this chapter, we study repair strategies for such conversational breakdowns, specifically in the context of spoken task-oriented conversations with a mobile device, grounded in the apps and the display of that mobile device. The new approach described in this chapter enables SUGILITE to better support users to effectively discover, identify the causes of, and repair breakdowns caused by certain natural language understanding errors.

Prior breakdown repair methods in human-agent conversations mainly have used only the natural language modality. Code-switching [283], which refers to the act of adjusting the speaking

This chapter is modified from the conference paper: Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M. Mitchell, and Brad A. Myers. Multi-Modal Repairs of Conversational Breakdowns in Task-Oriented Dialogs. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2020)*.

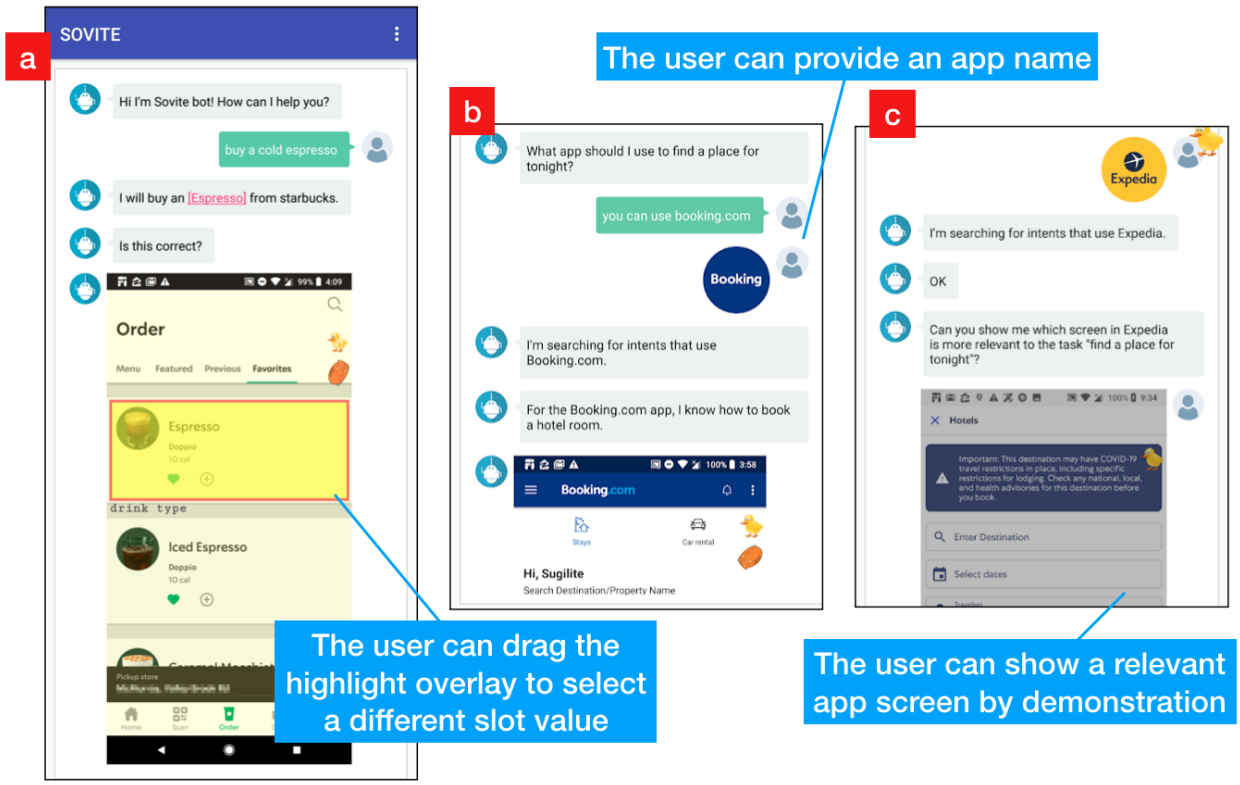


Figure 7.1: The interface of SOVITE: (a) SOVITE shows an app GUI screenshot to communicate its state of understanding. The yellow highlight overlay specifies the task slot value. The user can drag the overlay to fix slot value errors. (b) To fix intent detection errors, the user can refer to an app that 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.

style to accommodate the listener [19] in the context of conversational agents, is commonly found in users' breakdown repair attempts. Many repair strategies are used in these adjustments, depending on the user's understanding of the cause of the breakdown [19, 214]. For example, when the user suspects that the system has misheard the utterance, they might apply prosodic changes (adjust the rhythm or cadence of speech), overarticulation (exaggerating sounds), increased volume, or simply repetitions of the original utterance [19]. They might make syntactical adjustments to the original utterance if they thought the system did not understand its syntactic structure [19]. Similarly, they might perform semantic adjustments and modifications, such as replacing a word with its synonym, defining a concept, or breaking down a procedure, if they suspected the incorrect semantic understanding to be the cause of the breakdown [19, 214].

However, these strategies are often ineffective for two reasons: First, users often lack an accurate understanding of the cause of the breakdown because current conversational agents do not provide sufficient transparency into the system's state of understanding [19]. Understanding *why* a breakdown happens is crucial for the user to repair it [8]. In current agents, breakdowns are often discovered by users only after the system has acted incorrectly based on its misunderstanding (i.e., performing the wrong action) or from a generic error response (e.g., "Sorry I don't understand") [19]. Thus, little useful information is available for users to infer the cause of the breakdown. The system will provide a task-specific clarification response only in the small portion of cases where a developer has explicitly programmed an error handling conversation flow for the specific breakdown scenario.

Second, even when users correctly identify the causes of the breakdowns, their repairs in natural language are often ineffective [19]. This is especially problematic in breakdowns caused by natural language understanding errors, where the user needs to make semantic and syntactic adjustments such as verbally defining a keyword, explaining a procedure, replacing a word with its synonym, or restructuring the sentence instead of simply repeating the utterances with exaggerating sounds. The user usually needs to *guess* the right strategy to use due to the lack of transparency into the system's natural language understanding models. Existing conversational systems also often have problems understanding these repairs due to the system's limited capability of reasoning with natural language instructions and common sense knowledge [167]. This is part of the reason why commercial systems like Siri and Alexa rarely ask users to try to repair the conversation, but just immediately perform generic fallback actions such as searching the web.

Visualizing the agent's understanding of user intent is a promising way to address these challenges. Some existing agents support displaying visual responses (known as *cards*[1] in Google

[1] https://developers.google.com/assistant/conversational/rich-responses

Assistant) within natural language conversations. These cards allow the user to *see* the agent's state, and to interact with the agent visually to complement speech commands. However, these cards must be hard-coded by the developers, require significant effort to create, and therefore have limited adoption across task domains. In this chapter, we propose a new approach to address these challenges leveraging the graphical user interfaces (GUIs) of existing apps. This approach can be applied on devices with screens and access to app GUIs, such as smartphones and smart display devices (e.g., Amazon Echo Show, Google Home Hub). Our approach helps the user discover breakdowns and identify their causes by showing the system's state of understanding using GUI screenshots of the underlying apps for the task domain. It subsequently allows the users to repair the breakdowns by using direct manipulation on the screenshots and by making references to relevant apps and screens in the conversation.

The use of app GUIs has many advantages. GUIs and conversational agents are two different types of interfaces to mostly *the same* set of underlying computing services, since most, if not all, supported task domains in task-oriented conversational agents have corresponding mobile apps. The GUIs of these apps encapsulate rich knowledge about the flows of the tasks and the properties and relations of relevant entities [168]. The majority of users are also familiar with the app GUIs, the way to interact with them, and their general design patterns [163], which makes them ideal mediums through which the agent and the user can communicate in a supplementary modality during natural language conversations.

This chapter introduces SOVITE[2], 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 (Figure 7.1). A remote user study of 10 participants using SOVITE in 7 conversational breakdown scenarios showed that end users were able to successfully use SOVITE to fix common types of breakdowns caused by natural language understanding errors. The participants also found SOVITE easy and natural to use.

This chapter describes the following research contributions:

1. A new multi-modal approach that allows users to discover, identify the causes of, and repair conversational breakdowns caused by natural language understanding errors in task-oriented agents using the GUIs of existing mobile apps.

2. The SOVITE interface: an implementation of the above approach on SUGILITE, along with a user study evaluating its effectiveness and usability.

[2] 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.

7.2 Problem Setting

7.2.1 Frame-Based Task-Oriented Conversational Agents

SOVITE is designed for handling natural language understanding errors in task-oriented conversational agents that use the *frame-based* architecture for dialog management [30]. This is the most popular architecture used by the majority of existing commercial task-oriented agents [127]. It is based on a set of *frames* (also known as *intents*). Each frame represents a kind of intent that the system can extract from user utterances. A frame often contains *slots* whose values are needed for fulfilling the underlying task intent. For a user utterance, the system first determines the frame to use (e.g., finding a flight), and then tries to fill the slots (e.g., date, departure airport, and arrival airport) through the dialog. Frames can ensure the necessary structures and constraints for task completion [127], which are lacking in statistical dialog approaches (e.g., deep reinforcement learning for response generation [156, 157, 246] and information retrieval-based structures [290]) often used in social chatbots that converse with human users on open domain topics without explicit task completion goals.

A typical pipeline for a frame-based dialog architecture consists of multiple steps, including: (1) speech recognition, (2) natural language understanding, (3) natural language generation, and (4) speech synthesis [127]. Among those, (2) natural language understanding causes most of the critical breakdowns, as reported in Myers et al.'s study [214] on how users overcome obstacles in voice user interfaces. For the (1) speech recognition step, state-of-art algorithms have reached human parity [292]. For screen-based systems like what we use, the user can easily read and edit the transcription from the speech recognition step. Further, users' natural repair strategies such as repetition, prosodic changes, and overarticulation are effective for speech recognition errors [19]. The (3) natural language generation step in frame-based agents is typically rule-based with manually created templates, therefore less prone to errors compared with statistical approaches commonly used in social chatbots. Finally, the latest techniques for the (4) speech synthesis step reliably produce clear and easy-to-understand synthetic speech from text.

Problem Scope

This work, therefore, focuses on the breakdowns caused by (2) natural language understanding errors. There are two key components in the natural language understanding step: *intent detection* and *slot value extraction*. Intent detection errors are those where the system misrecognizes the intent in the user's utterance, and subsequently invokes the wrong dialog frame (e.g., responding "what kind of cuisine would you like" for the command "find me a place in Chicago tonight"

when the user intends to book a hotel room.) In slot value extraction errors, the system either extracts the wrong parts in the input as slot values (e.g., extracting "Singapore" as the departure city in "Show me Singapore Airlines flights to London.") or links the extracted phrase to incorrect entities (e.g., resolving "apple" in "What's the price of an apple?" to the entity "Apple company" and therefore incorrectly invoking the stock price lookup frame). As illustrated in the examples, slot value extraction errors can cause either the wrong slot match in the dialog frame (the airline example) or the wrong dialog frame (the apple example).

Note that there are other types of breakdowns in task-oriented agents that SOVITE does not handle. It does not address (1) speech recognition errors, as discussed previously. It also does not address breakdowns caused by errors in task fulfillment (i.e., exceptions when executing the task), errors in generating agent response ((3) and (4)), or the user's lack of familiarity with intents (e.g., the user does not know what intents the system can support).

Instructable Agents

An *instructable agent* like SUGILITE is a promising new type of frame-based agent that can learn intents for new tasks interactively from the end user's natural language instructions [10,258] and/or demonstrations [211], as discussed in previous Chapters. However, supporting effective breakdown repair is even more challenging for instructable agents. The user-instructed task domains have many fewer example utterances (usually only one) for training the underlying natural language understanding model. As a result, breakdowns caused by natural language understanding errors occur more frequently.

Further, when encountering breakdowns, the instructable agents seldom have task-specific error handling mechanisms for user-instructed task domains. In agents with professionally-developed task domains, developers can explicitly program error handling conversation flows for domain-specific breakdowns [5, 197]. However, end users with little programming expertise seldom create such error handlers for user-taught tasks due to the lack of tool support, and more importantly, the lack of expertise to consider and to handle possible future breakdown situations in advance.

We support instructable agents in SOVITE by (1) supporting user-instructed task domains by not requiring existing domain knowledge, hard-coded error handling mechanisms, or large corpus in the task domains, and (2) supporting the easy transition to user instruction when a breakdown turns out to be caused by out-of-domain errors rather than natural language understanding errors.

7.2.2 Design Goals

We have the following design goals:

1. The system should enable users to effectively discover, identify the causes of, and repair conversational breakdowns in intent detection and slot value extraction of frame-based task-oriented conversational agents.
2. The system should handle conversational breakdowns in various task domains, including *user-instructed* ones, without requiring existing domain knowledge or manually-created domain-specific error handling mechanisms.

7.3 Background

7.3.1 Studies of Breakdowns in Conversational Interfaces

The design of SOVITE is informed by insights from previous studies of breakdowns. As reported in Beneteau et al.'s 2019 deployment study [19] of Alexa, all study participants experienced breakdowns when conversing with the agent. The participants used a variety of repair strategies based on their understandings of the causes of the breakdowns, but those repairs were often not effective since their understandings were frequently inaccurate. Repairing breakdowns caused by natural language understanding errors (compared with speech recognition errors) was particularly problematic, as the natural repair strategies used by users such as semantic adjustments and defining unclear concepts were not well-supported by the agents. Other studies [8, 32, 56, 124, 214, 233] reported similar findings of the types of breakdowns encountered by users and the common repair strategies.

In a 2020 study conducted by Cho et al. [56], more than half of the responses given by Google Home in a user study with five information-seeking tasks were "cannot help" error responses (40%) or "unrelated" responses (24%) that were not useful for the user's request. In most cases, the "cannot help" messages did not provide useful information about the causes of the breakdowns to help the users with breakdown repairs. The participants were sometimes able to infer the causes of the breakdowns from "unrelated" responses, but this process was often unreliable and confusing. The 2018 study by Myers et al. [214] identified "Rely on GUI" as a strategy that users naturally use when they encounter obstacles in conversational interfaces, where they looked at the corresponding app's GUIs to look for cues on what to say for task intents.

Motivated by these insights, SOVITE helps the users identify the causes of breakdowns by visualizing the system's state of understanding of user intent using the app GUI screenshots.

7.3.2 Multi-Modal Mixed-Initiative Disambiguation Interfaces

SOVITE uses a multi-modal approach [222] that visually displays the system states and enables direct manipulation inputs from users in addition to spoken instructions to repair breakdowns, unlike prior systems that use only the natural language inputs and outputs [5, 8, 197]. SOVITE's design uses the *mutual disambiguation* pattern [221], where inputs from one modality are used to disambiguate inputs for the same concept from a different modality. Similar patterns have been previously used for handling errors in other recognition-based interfaces [190], such as speech recognition [263] and pen-based handwriting [141]. Visually-grounded language instructions were also used in interactive task learning for robots [248] and performing web tasks [3, 243].

The design of SOVITE also applies the principles of mixed-initiative interfaces [111]. Specifically, it considers breakdown repairing as a human-agent collaboration process, where the user's goals and inputs come with uncertainty. The system shows guesses of user goals, assists the user to provide more effective inputs and engages in multi-turn dialogs with the user to resolve any uncertainties and ambiguities. This combination of multi-modal and mixed-initiative approaches has been previously applied in the interactive task learning process in systems such as [3,137,163,167]. SOVITE bridges an important gap in these systems, as they focus on the ambiguities, uncertainties, and vagueness embedded in the user instructions of new tasks, while SOVITE addresses the conversational breakdowns caused by the natural language understanding problems when users invoke already-supported tasks.

In terms of the technique used, SOVITE extracts the semantics of app GUIs [53,183] for grounding natural language conversations. Compared with the previous systems that used the semantics of app GUIs for learning new tasks [163, 165, 247], extracting task flows [168], and supporting invoking individual GUI widgets with voice commands [269], a new idea in SOVITE is that it encodes app GUIs into the same vector space as natural language utterances, allowing the system to look up semantically relevant task intents when the user refers to apps and app GUI screens in the dialogues for repairing intent detection errors (details in Section 7.5).

7.4 The Design of SOVITE

7.4.1 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 Figure 7.1a, for the user command, SOVITE displays one or more (when there are multiple slots spanning many screens) screenshots from an app that correspond to the detected user intent (details of how SOVITE extracts the screenshots and creates the highlight overlays are discussed in Section 7.5). 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 Figure 7.1a, 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 if this is indeed the correct understanding of the user's intent by asking, "I will... `[the task]`, is this correct?", to which the user can verbally respond.

For example, as shown in Figure 7.1a, SOVITE uses the screenshot of the "Order" screen in Starbucks app to represent the detected intent "buy a `[drink type]` from Starbucks", and highlights the value "Espresso" for the slot `drink type` on the screenshot.

Design Rationale

SOVITE's references to app GUIs help with *grounding* in human-agent interactions. In communication theory, the concept of grounding describes the conversation as a form of collaborative action to come up with common ground or mutual knowledge [60]. 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 [39]. As described in the *gulf of evaluation* and *gulf of execution* framework [117, 219] and shown in prior studies of conversational agents [8, 19], 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.

The app GUI screenshots can be ideal mediums for communicating the state of understanding of the system. They show users the *evidence* of grounding [56] through their familiar app GUIs. This approach highlights that the agent performs tasks on the user's behalf, showing the key steps of navigating app screens, selecting menu items, and entering text through GUIs based on slot values as if a human agent was to perform the task using the underlying app. We believe this approach should help users to more effectively identify the understanding errors because it provides better *closeness of mapping* [99] to how the user would naturally approach this task. Prior studies showed that references to existing app GUIs were effective in other aspects of conversational interface

designs, such as enabling users to explain unknown concepts [167] and helping users come up with the language to use to invoke app functionalities [214]. Showing the screenshots of GUIs is also a useful way to present the context of a piece of information, making the content easier to understand [182].

7.4.2 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 Figure 7.1b). 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 (we will discuss how SOVITE finds semantically relevant task intents in Section 7.5).

For example, suppose the agent displays the screenshots of the OpenTable app for the utterance "find a place for tonight" because it classifies the intent as "book restaurant" when the user's actual intent is "book hotel". The user notices that the app is wrong from the screenshot and responds "No" when the agent asks, "Is this correct?" The agent then asks which app to use instead, to which the user answers "Booking.com", which is an app for booking hotel rooms. If there was indeed a supported "book a hotel room" intent using the Booking.com app, SOVITE would respond "OK, I know how to book a hotel room using Booking.com", and then show the screenshots of booking hotel rooms in Booking.com for confirmation. If no such intent was available, but there was an intent semantically related to the Booking.com app, such as a 'book a hotel room" intent using the Hilton app, SOVITE would respond "I don't know how to find a place for tonight in Booking.com, but I know how to book a hotel room using Hilton" and show the corresponding screenshots. The user can verbally confirm performing the task using the Hilton app instead, or indicate that they still wish to use the Booking.com app and teach the agent how to perform this task using Booking.com by demonstration.

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 Figure 7.1c). SOVITE then launches the app and asks the user to navigate to the target screen in the app. (The user may also say "no" and start over.) After the user reaches the target screen, they can click on a floating SOVITE icon (the duck) to provide this screen as an input to SOVITE. SOVITE then finds intents that are the most semantically related to this app screen among the ambiguous ones (technical details in Section 7.5), 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 *instructing* how to perform a new task if the intent disambiguation attempt fails when the user's intended task is out of scope (i.e., there is no existing intent that supports the task). SOVITE's approach can directly connect to the user instruction mode in SUGILITE. Since at this point in the overall process, 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 (technical details in Section 7.5).

7.4.3 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 (Figures 7.1a and 7.2). For slots represented by GUI menu selections, the user can simply drag the highlight overlay to select a different item. For example, assuming the agent incorrectly selects the item "Espresso" for the utterance "order a cold espresso" due to an error in entity recognition, as shown in Figure 7.1a, the user can drag the highlight overlay to "Iced Espresso" on the screenshot to specify a different slot value. (If the user's desired slot value is not on the screen, the user can say "no" and indicate the correct screen to use, as discussed in the previous section.) 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 (Figure 7.2). When a field is dragged to another field that already has a value, SOVITE performs a *swap* rather than a *replace*ment 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 Figure 7.2. 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 (e.g., the system recognizes the slot value as "The Lego" when the user says "find showtimes for *The Lego Movie*.") 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—this is also supported by the results reported in Section 7.6. 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

Figure 7.2: 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.

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 [251].

7.5 Implementation

We implemented SOVITE in Java as an Android app. SOVITE was developed and tested on a Google Pixel 2 XL phone running Android 8.0. It does *not* require root access to the phone and should run on any phone with Android 6.0 or higher.

The implementation of SOVITE builds on the SUGILITE system described in this dissertation. We summarize SUGILITE's key characteristics relevant to SOVITE below. You can refer to Chapter 3 and Chapter 5 for more details.

SUGILITE uses a standard frame-based dialog management architecture with intents, slots, and slot values. Its natural language model uses a SEMPRE [24]-based Floating Parser [227] that can parse the user's utterance into a corresponding expression that invokes an intent and sets the respective slot values. The model was trained on the lexical (e.g., unigrams, bigrams, skip-grams),

syntactic (e.g., part-of-speech tags, named-entity tags), and semantic (e.g., word embeddings) features extracted from the training utterances for each task intent, including when users teach new tasks for an utterance (details in Chapters 4 and 5). Because the task fulfillment in SUGILITE is instructed by end users, there are usually only a very small number of sample training utterances (often only one) for each task intent. As a result, conversational breakdowns are common in SUGILITE's interaction with users when they use utterances with diverse vocabulary, structures, or expressions that are not covered in the training corpus.

While we implemented SOVITE with our SUGILITE agent, we believe the approach used in SOVITE should generalize to other frame-based task-oriented conversational agents as well. The only major part of SOVITE's implementation that is specific to SUGILITE is its mechanism for generating app GUI screenshot confirmations. However, there are other practical ways to generate these app GUI screenshot confirmations without relying on the programming by demonstration scripts in SUGILITE (details in the next section).

7.5.1 Generating the App GUI Screenshot Confirmations

In SUGILITE, each supported task intent corresponds to an automation script created from user demonstrations of performing the task manually using the GUIs of the underlying app. Therefore, SOVITE can extract app GUI screenshots for these intents by instrumenting the demonstration process.

When the user starts demonstrating a task, SOVITE creates a virtual display device in the background that mirrors the main display that the user sees for capturing the screenshots. For each GUI action demonstrated by the user, SOVITE takes a screenshot that captures this action. At the end of the demonstration process, SOVITE compares the task slot values with the demonstrated actions to identify actions that correspond to the task slots and saves these screenshots to be used as the confirmation for this demonstration's underlying task intent. For example, assuming the user's demonstration for the task "order an Espresso" contains an action "click on the item 'Espresso"' from a menu on the GUI of Starbucks app, SOVITE will use the screenshot taken from the user demonstrating this action as a confirmation for the intent. The same mechanism also works for slot values from text inputs (e.g., the user demonstrates typing "Chicago" into a textbox for a command "book a hotel room in Chicago").

Although the implementation of generating app GUI screenshot confirmations used in SOVITE, as described above, only applies to programming-by-demonstration instructable agents such as SUGILITE [159], PLOW [3], and VASTA [247], there are other feasible approaches for generating app GUI screenshot confirmations in other types of agents. For example, recent advances in ma-

chine learning have been shown to support directly matching natural language commands to specific GUI elements [226] and generating semantic labels for GUI elements from screenshots [53]. For agents that use web API calls to fulfill the task intents, it is also feasible to compare the agent API calls to the API calls made by apps by analyzing the code of the apps (e.g., CHABADA [97]), or to the network traffic collected from the apps (e.g., MobiPurpose [125]). These techniques should allow associating slots with their corresponding app GUI widgets without relying on user demonstrations.

7.5.2 Finding Relevant Intents from Apps and App Screens

When the user refers to an app name or an app screen for their desired task for disambiguating task intents, SOVITE first looks for intents that use exactly this app or this app screen. If none of the supported task intents uses the exact app or app screen that the user refers to, SOVITE can recommend relevant task intents. For example, if the user refers to the app "Booking.com" or the page for "List hotels near `[location]`" in Google Maps to explain the utterance "find a place for tonight", SOVITE can prompt "I know how to book a hotel room using the Hilton app, is this what you want to do?" (assuming that the underlying SUGILITE agent has been previously taught the skill of "book a hotel room" using the Hilton app)

The technical challenge here is to identify semantically relevant task intents based on the user-provided app names and app screens. An effective way to match the user's task intent with natural language descriptions of goals (e.g., book hotel) to apps (e.g., Booking.com) is to leverage the app descriptions in the app stores. For example, MessageOnTap [52] uses word embeddings to represent the semantic meanings of individual words in the user utterances and app store descriptions and calculates the cosine similarity between the word embedding centroids of each app description and the conversation to recommend relevant apps in human-human conversations (e.g., recommending the Calendar app and the OpenTable app when one party in the conversation says "let's schedule a dinner.") SOVITE uses a similar approach but recommends task intents from the user references to apps instead of recommending apps from the user expressions of task intents as in MessageOnTap.

Specifically, for each app reference made by the user, a remote SOVITE server retrieves its description from the Google Play store and calculates the sentence embeddings for the app store description and the sample utterances of each supported task intent using a state-of-art pre-trained Sentence-BERT model [238], which is a modified BERT network [69] that can derive semantically meaningful sentence embeddings for calculating semantic relatedness. SOVITE is then able to identify the most relevant task intent for the app by finding the task intent whose centroid of

#	Breakdown Type	Example Scenario	User Repair Method with SOVITE
1	No error	Make a call to Amazon (in the Phone app)	Not applicable
2	Intent: no intent matched (did not understand the command)	Find something to eat (should order food for delivery in Doordash)	Provide a reference to the correct app to use (and the screen if needed)
3	Intent: wrong app used	Find a place for tonight (recognized as using OpenTable for restaurant booking instead of using Hilton for hotel booking	Provide a reference to the correct app to use (and the screen if needed)
4	Intent: correct app, wrong screen used	Book a ticket to New York (recognized as booking a hotel room in Expedia instead of booking a flight)	Provide a reference to the correct screen in Expedia to use
5	Slot: wrong item selected in a menu	Buy an Iced Espresso from Starbucks (the slot value recognized as "Espresso")	Drag the highlight on the screenshot to select the correct item
6	Slot: wrong value extracted for text input	Find the showtimes of The Lego Movie (the slot value recognized as "The Lego")	Click on the highlight on the screenshot to modify the slot value
7	Slot: slot value mismatched	Book an Uber ride to airport from home (the starting location and the destination are swapped)	Drag the highlight on the screenshot to swap slot values

Table 7.1: The 6 breakdown types and a "no error" type covered in the user study, an example scenario for each type, and their corresponding user repair methods using SOVITE. All participants saw all 7 in random order.

sentence embeddings of all its sample utterances has the highest cosine similarity with the sentence embedding of the target app's app store description. All this can be done in real-time as the user is interacting with SOVITE.

When the user refers to a specific app screen, SOVITE uses a similar technique for finding semantically relevant task intents. The only difference is that instead of using the sentence embeddings of app store descriptions, it uses the embeddings of all the text labels shown on the screen, and computes their semantic relatedness with each supported task intent to find the most relevant one.

7.6 User Study

We conducted a remote user study[3] to evaluate SOVITE. The study examined the following two research questions:

RQ1: Can users understand and use SOVITE's new features for identifying and repairing conversational breakdowns?

RQ2: Is SOVITE effective for fixing conversational breakdowns caused by natural language understanding errors in task-oriented agents?

7.6.1 Participants

We recruited 10 participants (2 identified as female, 8 identified as male, ages 25-41) for our study. 5 participants were graduate students in two local universities, and the other 5 worked at different technical, administrative, or managerial jobs. All of our participants were experienced smartphone users with more than 3 years of experience of using smartphones. 8 of the 10 participants (80%) were active users of intelligent conversational agents such as Alexa, Siri, or Google Assistant. Each participant was compensated $15 for an hour of their time.

7.6.2 Study Design

The remote study session with each participant lasted 30–40 minutes. After agreeing to the consent form presented online and filling out a demographic survey, each participant received a short tutorial on SOVITE, which explained the SOVITE features discussed above. The participant was then presented with the 7 tasks in random order. In each task, the participant saw an example conversation scenario that contained a user voice command and SOVITE's response (i.e., each scenario tells the participant "Assume you have said `[utterance]`, and here is the agent's response. You need to identify whether the system's understanding of the intent was correct and fix the breakdown using SOVITE when the understanding was incorrect"). The 7 tasks include one "no error" scenario (Scenario 1), and 6 breakdown scenarios that cover different types of intent detection and slot value extraction errors (Table 7.1). The participant then filled out a post-study questionnaire about their experiences with SOVITE, and ended the study with a short interview with the experimenter.

The study was performed remotely using the Zoom video conference software. SOVITE ran on a Pixel 2 XL phone running Android 8.0 with relevant third-party apps pre-installed. We streamed

[3]The study protocol was approved by the IRB at our institution.

the screen display of the phone through a camera pointing at its screen, so that the remote participant could see and hear the output of the phone. The participant was able to control the phone *indirectly* through the relay of the experimenter: the participant could point to a GUI widget on the phone screencast with the mouse cursor, and ask the experimenter to click on the widget or enter text into the widget on their behalf. The participant could also ask the experimenter to say something by speech to the phone, and the experimenter repeated the participant's utterance exactly. Since speech recognition errors were not a concern for this study, repeating the utterance was not a confound.

Impact of the COVID-19 Pandemic

This study was conducted in April 2020 in the midst of the COVID-19 global pandemic. Due to health concerns, we were unable to conduct an in-person lab study as originally planned. Although in the remote study the participants were not able to directly control and speak to SOVITE, we believe the results were still valid for the two research questions we asked. Specifically, the study measured whether the users could come up with what to do in SOVITE when presented with breakdown situations and whether these inputs were effective for breakdown repairs. We tried a few third-party software tools for directly screencasting and remote controlling Android phones but ran into stability and performance issues with them when sharing with remote participants; therefore we used the camera method described above.

Note that this is *not* a Wizard-of-Oz study because the system was actually operating on the participants' utterances and actions—the experimenter just served as an intermediary since we could not have the participants use the actual phone.

7.6.3 Results

Among the 70 scenario instances (10 participants $\times$ 7 scenarios), including 10 "no error" scenarios, the participants correctly identified all 10 "no error" scenarios and discovered 57 out of 60 errors (95%). Among the discovered errors, they successfully fixed *all* of them using SOVITE. The participants failed to notice the error in two instances of Scenario 7 and one instance of Scenario 6. When asked to reflect upon their experience, the participants attributed all of these failure cases to the "expectation of capabilities" problem, which we will describe in Section 7.7 below.

Subjective Results

In a questionnaire after the study, we asked each participant to rate statements about SOVITE's usability and usefulness on a 7-point Likert scale from "strongly disagree" to "strongly agree". SOVITE scored on average 6.1 ($SD = 0.83$) on "*I find* SOVITE *helpful for fixing understanding errors in conversational agents*", 6.4 ($SD = 0.8$) on "*I feel* SOVITE *is easy to use*", and 6.3 ($SD = 0.9$) on "*I'm satisfied with my experience with* SOVITE" Specifically for SOVITE's individual features, the participants rated 6.5 ($SD = 0.81$) on "*The highlights on screenshots for task parameter values are clear*" and 6.2 ($SD = 1.54$) on "*Dragging the highlights to fix parameter errors is natural.*" These results suggest that our participants were positive about the usability and usefulness of SOVITE.

7.7 Discussion

We observed some confusion over the highlight overlays in screenshot confirmations in the participants' interactions with SOVITE. A few participants tried to interact with the other GUI components on the screenshots when they first encountered them. For example, in Scenario 4, P1 tried to use the back button on the screenshot to switch to the "book flight" page. In Scenario 7, P2 was looking for a "swap" button on the screenshot. However, after trying to interact with these GUI components and receiving no response, they quickly realized that only the highlight overlays were interactive on the screenshots and successfully repaired the breakdowns thereafter. The overlay is not a common metaphor in interfaces—users are more familiar with static screenshots where nothing is interactive, and actual GUIs where every element can be directly interacted with. The overlay (also known as *interaction proxy* [304]) sits in between, where the user can specify actions *about* an underlying GUI element (e.g., the value that *should go into* a textbox or the menu item that *should be selected*) using direct manipulation, but not directly interact with these GUI elements. A future direction is to explore the design space of overlay interfaces to make them more intuitive to use.

The "expectation of capabilities" problems [8, 19, 214] impacted the user's capability to discover errors in SOVITE in some cases. For example, the visual clue for Scenario 7 was rather subtle on the screenshot (i.e., the highlights for starting location and destination in Uber were swapped, as shown in Figure 7.2). P7 missed this error and went with "OK" when SOVITE asked for confirmation. When asked about this instance after the study, P7 said "*I didn't expect it [the system] to make this error... I thought the original utterance was clear, so when I saw two highlights saying "home" and "airport" I didn't carefully check the order since I assumed that the system would get it right... I was more looking at if this was indeed the request ride screen in Uber.*" Users may

look more carefully at places where they expect errors to appear, but their expectations might not always match the system's behaviors. However, with SOVITE, once the user discovers an error, it is straightforward what the cause of the error is and how to fix it, which is a significant improvement from the prevailing systems where the user needs to guess the cause of the error (which is often inaccurate) and come up with the repair strategy to use (which is often ineffective as a result) [19, 214].

Efficiency wise, we did not measure the time-on-task in the study due to the delays and overheads from running the study remotely. But from our observations, while adding some overhead to the conversation, using SOVITE should still be more efficient than completing the tasks manually in most cases. For example, completing the "order coffee" task in Starbucks requires up to 14 clicks on 8 screens. In comparison, the user reads one GUI screenshot confirmation (and fixes the errors, if any) in addition to speaking the initial utterance when using SOVITE for the same task.

Design Implications

SOVITE illustrates the effectiveness of presenting the system's state of understanding in a way closely matched to how the user would otherwise (i.e., not using speech) naturally approach the problem to help with error discovery. While speech is a natural modality for interacting with task-oriented agents, the technical limitations in natural language understanding and reasoning capabilities limit its effectiveness in handling conversational breakdowns. App GUI screenshots can serve as a good complement to natural language in this context.

An important underlying assumption in SOVITE's strategy of using app GUI screenshots is that users are familiar with the app GUIs. This is also the assumption of many prior interactive task learning systems like [3, 152, 159, 247]. SOVITE requires the user to have a mental model of "apps" so that they understand how to complete their intended tasks through existing app GUIs. Today, this assumption seems reasonable, given the high adoption rate of smartphones [45]. Most app GUIs are designed to be easy-to-use with common design patterns [67], so users are likely able to understand the screens even if they have never used the particular app before. Using app GUIs is still by far the most common means through which the users access computing services. However, it would be interesting to think about how this may change for certain user groups in some task domains in the future. For example, are we going to see the conversational agents become "the native interface" for some tasks and some user groups in the future, just as how GUIs replaced command-line interfaces? In our opinion, the app-GUI-based approach used in SOVITE can be a stepping stone to a more integrated speech-oriented agent in the future, which may eventually transcend the app-oriented design of current smartphones.

SOVITE's design highlights the importance of *consistency* between how users fix the errors and how they discover the errors. Once the users discover the errors (e.g., the wrong app was used, the wrong screen was shown, the highlights are at wrong places, the slot values use the wrong parts of the initial utterances, etc.), the ways to fix them are rather intuitive and obvious (e.g., saying the correct app, pointing to the correct screen, dragging the highlights to the correct places, and selecting the right parts of the initial utterances to be used in the slot values). This was noticed and praised by many participants in our study. With SOVITE, the user no longer needs to guess the strategy to fix the error (e.g., explain a word, replace words with synonyms, restructure the syntax) like with prevailing systems when the error message was the generic "Sorry I didn't understand".

7.8 Limitations and Future Work

SOVITE currently does not handle task intents that span multiple apps (even though the underlying SUGILITE system does). Those intents are often higher-level intents that involve multiple smaller sub-intents within individual apps and information exchange between them (e.g., "plan a dinner party" can involve "find time availability for `[people]` in Calendar", "make a restaurant reservation at `[time]`", and "notify the `[people]` about the `[reservation info]`". An interesting future challenge is to design a new confirmation mechanism to clearly show the system's state of understanding for such cross-domain intents with increased complexity.

As discussed previously in Section 7.2.1, SOVITE only handles intent detection and slot value extraction errors in natural language understanding. We hope to integrate SOVITE with the existing mechanisms that handle other kinds of errors, such as voice recognition errors, task execution errors (described in Section 3.3.3), and feedback generation errors.

SOVITE currently displays the full app GUI screenshots as a part of the conversation. As a result, the screenshots are displayed in about half of their original size, making them harder to read for the users. The highlight overlays for smaller GUI elements are also prone to the "fat finger" problem. To address this issue, one approach is to extend our model for extracting app GUI screenshots so that it only displays parts of the screens that are most relevant to the underlying task intents. This might be feasible as we already have a mechanism to determine the semantic relatedness between GUI screens and task intents, but it risks making it harder for users to understand the context of the displayed portions. Another approach is to add support for zooming and panning using familiar gestures such as pinch-to-zoom on the screenshots.

Another challenge is to better encode the semantics of app GUI screens for assisting natural language understanding. SOVITE's current mechanism only takes the text labels shown on GUIs

into consideration. We can capture more comprehensive semantics of app GUI screens by leveraging the GUI layouts (e.g., the distance between elements [169] and design patterns [67, 183]), control flows among GUI screens [168], and large collections of user interaction traces. The availability of large-scale GUI datasets like RICO [66] makes future experiments in this area feasible. Chapter 9 discusses some of my work in this direction.

Lastly, the user-provided repairs in SOVITE only apply locally to the current dialog session. In the future, we plan to develop mechanisms that allow the conversational agent to *learn* from the user-provided repairs to improve its performance. The user's expression of the actual intent for their natural language command collected through SOVITE can be a highly valuable resource for applying *online learning* (a machine learning approach that supports incremental learning using small batches of data) to improve the accuracy of the agent's natural language understanding models on the fly.

7.9 Chapter Conclusion

Conversational breakdown repairing in task-oriented dialogues is surprisingly little studied or handled by research or commercial intelligent agents. The lack of effective and robust breakdown repair mechanisms significantly affects the adoption of these agents. By incorporating the SOVITE interface, the SUGILITE system is able to better support users to effectively discover, identify the causes of, and repair breakdowns caused by intent detection errors and slot value extraction errors.

Chapter 8

Privacy in Sharing Demonstrated Scripts

8.1 Introduction

An important focus of this dissertation is on end-user development (EUD). The concept of EUD, according to a popular definition, refers to programming activities to achieve results primarily intended for personal, rather than public use [138]. It allows individual users to develop and adapt systems according to their own needs and preferences [179]. Despite the "for-self-use" main intention, *sharing* of the program artifacts is common in EUD (e.g., [186]). However, when sharing EUD artifacts, users are often concerned about privacy: they do not want to leak any personal information that may be embedded in the shared artifacts [152, 155].

This concern is particularly important in graphical user interface (GUI) based programming by demonstration (PBD) systems such as SUGILITE. In such systems, end-user developers do not directly author the programs, but instead, demonstrate one or more examples of the desired system behaviors using the GUIs of target applications. From the demonstrations, the system synthesizes a program that can invoke and control the target applications to perform tasks. As a result, end-user developers have little knowledge or control about what has been included in the resulting program, and therefore are hesitant about sharing them [31, 152, 155]. In the recent GUI-based PBD agents for task automation (e.g., SUGILITE [159] and VASTA [247]), the system not only collects information about the exact demonstrated actions but also records rich contextual information (e.g., the contents of the screens including the app's responses) in order to better infer the users' intents from the demonstrations and to further generalize the resulting programs. While these new mechanisms make the PBD more powerful, flexible and robust, they also increase

This chapter is modified from the conference paper: Toby Jia-Jun Li, Jingya Chen, Brandon Canfield, and Brad A. Myers. Privacy-Preserving Script Sharing in GUI-based Programming-by-Demonstration Systems. In *Proceedings of the ACM on Human-Computer Interaction (CSCW 2020)*.

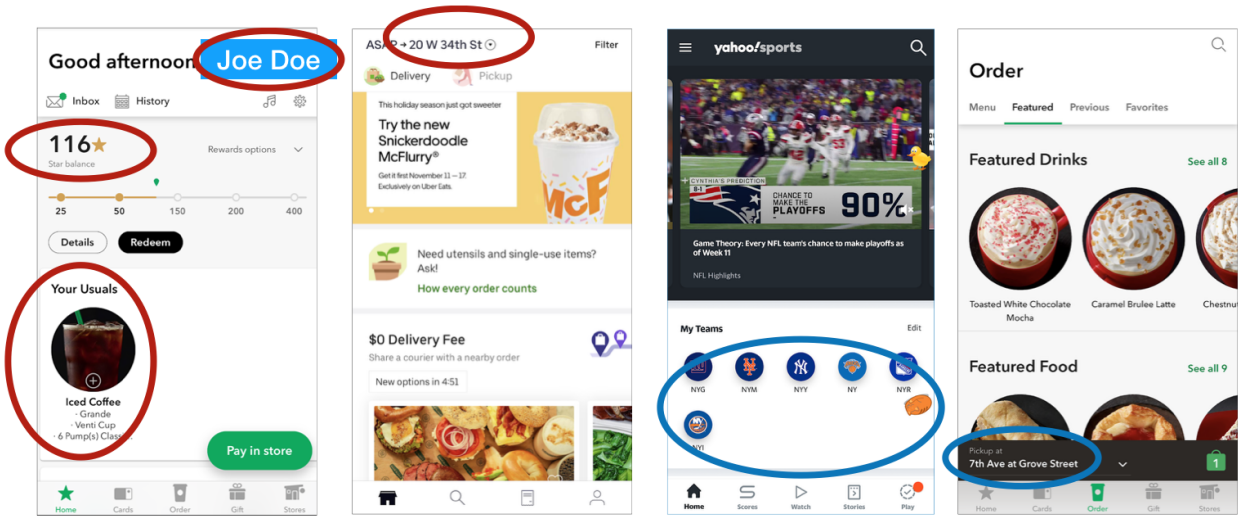


Figure 8.1: Some examples of potential information leaks from the app GUIs. The red circles highlight personal information directly displayed (e.g., name, point balance, order history, current address). The blue circles highlight information that can be used for re-identification attacks (e.g., the user's favorite teams and the name of a nearby Starbucks location can be used for inferring the user's location).

the likelihood of inappropriately including personal information in program artifacts, and make it infeasible for end-user developers to manually scrutinize the program artifacts before sharing them, since the collected contextual meta information is large in amount (averaging over 100 strings per script in our tests) and not generally shown to users. Many information fields displayed in app GUIs are dynamically generated based on the user's personal information. Figure 8.1 shows some examples of personal information displayed in app GUIs that may be included in PBD scripts.

Another constraint in removing personal information from program artifacts created from GUI demonstrations is to preserve the scripts' transparency, readability, robustness, and generalizability. In task automation, it is important to enable users who plan to use a script shared by others to be able to fully understand its behavior, so that they can (1) validate if the script actually fulfills their needs, (2) identify any potentially dangerous or malicious operations in the script, and (3) modify or extend the script when needed. As a result, the system should avoid unnecessary obfuscation of the shared scripts that hinders transparency and readability. The system should also preserve as much meta information from demonstrations as possible in shared scripts, since such meta-information is very useful for recovering from script execution errors, extending existing scripts, and supporting effective parameterization in scripts (see more details in Section 8.3.2).

There is also the emerging challenge of preventing *re-identification attacks* [21, 77] when protecting identifiable personal information. In many cases, user identities and private information can be re-identified by combining multiple seemingly innocent data sources. For example, knowing

seemingly non-private information like the user's local sports teams, the default Starbucks store, and the estimated wait time for food delivery, one can infer the user's location fairly accurately (Figure 8.1 shows some examples). This is also known as a *record linkage* attack [278]. Prior approaches that mask only pre-specified types of explicit personal information (e.g., [62, 152, 285]) are not sufficient when dealing with re-identification attacks.

In this chapter, we present a new mechanism named PINALITE[1] for privacy-preserving sharing of GUI-based PBD task automation scripts in SUGILITE. PINALITE's approach can identify and obfuscate possible personal information at the time of sharing with minimal user intervention. This approach collects and aggregates hashed text strings from app GUIs on users' phones in everyday use. This aggregated data allows our system to identify fields in end-user-developed scripts with potentially personal information, and obfuscate their values by hashing. Unlike prior systems, PINALITE works with third-party apps from any task domain without prior knowledge about the app or the domain, and can process broad types of personal information beyond explicitly pre-specified ones to protect against re-identification attacks. At runtime, the obfuscated fields can be rebuilt locally using the script consumer's app GUIs, which helps preserve the shared scripts' transparency, readability, robustness, and generalizability.

8.2 Related Work

8.2.1 Sharing and Cooperations in End User Development

The sharing of PBD scripts has been discussed in prior work. Leshed et al. outlined two main motivations for sharing end-user-developed PBD task automation scripts in CoScripter [152]: *(i)* for tasks that are tedious and are performed frequently by many users, the main motivation is efficiency; and *(ii)* for tasks that are complex or hard to remember, the shared scripts are also used as the medium for sharing *how-to* knowledge among users. A later field study of CoScripter scripts [31] revealed that users had shared CoScripter scripts for automating a wide range of work-related and non-work-related tasks, which affirmed the need for sharing end-user-developed scripts. The same study also confirmed that making scripts that were mainly intended for self-use be publicly available can lead to serendipitous reuse by others [31].

The privacy concern in PBD script sharing has been reported by several studies. Email and interview studies for CoScripter [152] showed that the privacy concern was a main barrier to sharing. Participants reported that they decided to make scripts private because they worried about having

[1] PINALITE is named after a type of rock. It is also an acronym for **P**ersonal **I**nformation **N**icely **A**nonymized **L**everaging **I**nterface **T**race **E**xamples.

personal information embedded in scripts. Studies and discussions for other PBD systems such as ActionShot [155], RePlay [86], and SUGILITE [159] also identified similar privacy concerns. Another prior study investigated the privacy risks of running trigger-action EUD scripts for the script users [267]. Our work focuses on the privacy risks of sharing PBD scripts for the script authors.

A few prior approaches have tried to address the privacy issue in sharing PBD scripts. CoScripter [152] used a "personal database" where end user developers could manually enter their personal information such as name, phone number and email address. Whenever these entries appear in the recorded scripts during demonstrations, they are replaced with named variables. During execution, these named variables are populated with data from the script consumer's personal database. A significant drawback of this approach is that it requires users to manually create personal databases, which requires significant effort. Personal databases also only cover the most common and the most obvious types of personal information, and only handle situations where database entries show up as exact string matches in the shared scripts. Another approach is to display operations in a script as a list of textual steps (e.g., ActionShot [155]) or in visual programming blocks (e.g., Rousillon [50] and Helena [49]) before sharing, and have users check if any personal information is included. This approach also requires manual intervention from users, and is not scalable to recent PBD systems [159, 164, 167, 247] that collect a large amount of contextual information along with the operations. In contrast, PINALITE seeks to minimize user effort, to cover a wide range of explicit and implicit personal data, and to support processing GUI contextual meta information embedded in automation scripts.

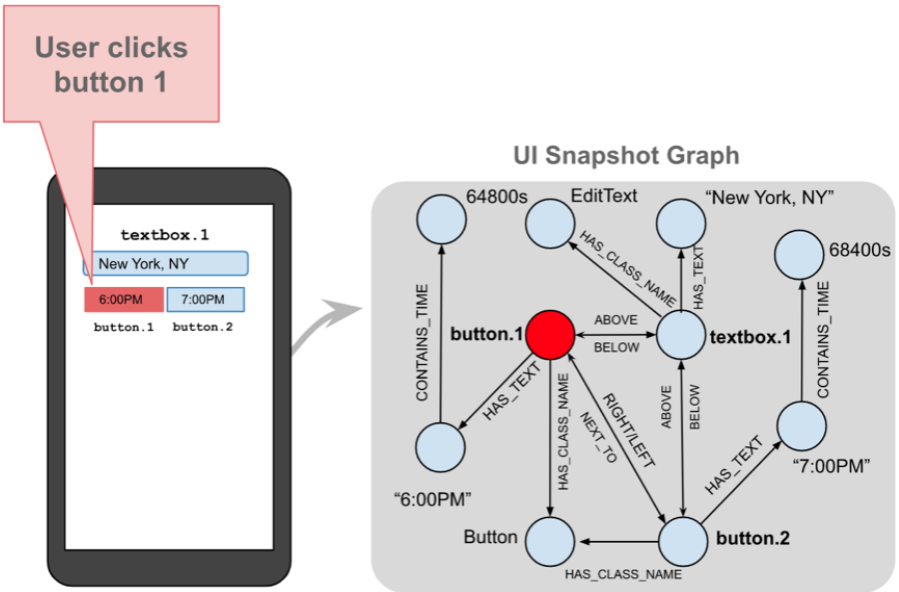
The tailoring of software artifacts is another important aspect of cooperative EUD. Some prior work in this area focused on the cooperation between end users, and sometimes professional developers or "local experts" during the development process (e.g., [90, 186, 216, 291]), which we do not consider in this chapter. In the scope of this chapter, we assume the development is done individually. Pipek and Kahler proposed three levels of intensity of user ties in collaborative tailoring ("shared use", "shared context", and "shared tool") [232]. The scenarios discussed in this chapter fall into the "shared use" level, where "the common denominator for cooperation" is the tool usage itself [232]. Each shared script can automate a type of task by invoking and controlling an app (or multiple apps) that both the script author and the script consumer have. The script author and the script consumers do not share the same task instances, the same workspace, or the same context, nor is there any data exchange between them besides the script itself.

8.2.2 Identifying Personal & Private Information in Data Sharing

Our approach is also related to prior literature from the privacy and security research communities on detecting personal information leaks in shared data. For example, systems like PrivacyProxy [257], Agrigento [62], MobiPurpose [125], and LeakDoctor [285] detect leaks of personal information by analyzing the contents of outbound network packets. Agrigento [62] and LeakDoctor [285] used black-box differential analysis, where they test if the outbound network traffic changes when values for personal information of interest (e.g., user id, location) are changed. This approach will work with any app, but can only protect a list of pre-specified common types of personal information (e.g., location, user id), not the non-obvious ones we discussed in Section 8.1 (e.g., the user's reward point balance and the user's order history as shown in Figure 8.1). TaintDroid [78] labels data from privacy-sensitive sources (e.g., GPS, contacts) and tracks how they flow within apps and leave the system, which also only works with limited pre-specified common types of personal information. Systems like SUPOR [114] detect sensitive information from user text inputs in mobile apps using natural language processing (NLP) techniques. This approach also only deals with pre-specified common types of personal information. Those methods are limited when dealing with the long-tail of uncommon types of personal information, and can be prone to re-identification attacks [21, 77]—the way to identify the user by combining multiple pieces of seemingly less-sensitive information.

PINALITE's approach for identifying personal information leaks is similar to the approach used in PrivacyProxy [257] – both systems identify potential personal and private information based on the uniqueness of the data. This approach does not limit the types of personal information that can be detected, providing coverage for more types of personal information. Compared with PrivacyProxy, which identifies private information in network traffic packets, PINALITE uses a new approach of estimating uniqueness for information shown on app GUIs (more details in Section 8.4.4). PINALITE also rebuilds obfuscated information at runtime on the script consumer's side using GUI information from the script consumer's local apps (see Section 8.4.6).

Another way to protect personal information is to monitor when, how, and what granularity of personal information is accessed and shared by apps [170]. This approach requires instrumenting each individual app, which is not scalable for our use case because we want to handle PBD scripts made with any arbitrary third-party app. In contrast, since PINALITE focuses on information extracted from the GUIs of apps, it does not require any special modification on target apps, and should work with most native Android apps (with a few exceptions, such as apps with custom graphic engines like games).



Example possible data description queries for button.1:

```
(AND (HAS_TEXT "6:00PM") (HAS_CLASS_NAME Button))
//the Button that has text "6:00pm"

(ARG_MIN CONTAINS_TIME (AND (HAS_CLASS_NAME Button)
       (BELOW (AND (HAS_CLASS_NAME EditText) (HAS_TEXT "New York, NY"))))
//the Button that has the earliest time below the EditText that has text "New York, NY"

(AND (NEXT_TO (AND (HAS_TEXT "7:00PM") (HAS_CLASS_NAME Button))) (HAS_CLASS_NAME Button))
//the Button that is next to the button that has text "7:00pm"
```

Figure 8.2: A simple example GUI, its corresponding UI snapshot graph, and some example possible data description queries for `button_1` on the graph (highlighted in red).

8.3 Background

The implementation of PINALITE builds on the SUGILITE system described in this dissertation. This section summarizes the key characteristics of SUGILITE scripts and their corresponding metadata, and the possible types of privacy leaks in them, providing the background for the design, development, and evaluation of our approach.

8.3.1 SUGILITE Scripts

SUGILITE scripts support task automation using the GUIs of Android mobile apps (e.g., to automate ordering coffee using the Starbucks app). Each script consists of a collection of operations, conditionals, and their corresponding metadata. In most cases, an operation represents an action to be performed on the GUI of an app, such as `CLICK` (click on a GUI element *X*) and `SET_TEXT` (set the value of the `text` field of a GUI element *X* to *Y*). For those operations, the first arguments (*X* in the previous two examples) are used for identifying the target GUI elements from current

screens for performing the actions (more details in Section 8.3.2). Some operations also have additional arguments (e.g., the second argument of a `SET_TEXT` operation specifies the String content that the value of the target GUI element's `text` field should be set to). There are also a few types of special operations that do not represent actions on the GUI, such as `READ_OUT` (read the `text` field of a GUI element out loud), `EXTRACT_VALUE` (assign a variable using the value of the `text` field of a GUI element), and `PAUSE` (pause for *X* seconds or until the user signals to continue). A script can have parameters (e.g., the *size* of coffee and the *type* of coffee for a coffee-ordering script). Each parameter may have a list of possible values (e.g., "tall", "grande" and "venti" for *size* in the Starbucks app). The parameters and their possible values are extracted from GUIs during the demonstrations, as explained in Chapter 3.

Operations are linked and executed sequentially unless there is a conditional. A conditional `IF` includes one or two (with an `ELSE` clause) branches and a Boolean expression. At runtime, the execution depends on the evaluation result of the Boolean expression, which can involve constants or variables. The values of variables can be either provided before the script execution, during the script execution through extracting values from GUIs, or from the result of executing a subscript. (see Chapter 5 for full details on conditionals and variables in SUGILITE scripts)

8.3.2 Data Descriptions and GUI Snapshot Graphs

In programming by demonstration, the main challenge is to infer the user's intent as explained in Chapter 4. From a recorded action, the system needs to determine the user's intent in performing that action so that the system can perform the desired action in a different context in the future. This process is known as creating a *data description* [65, 175]. For GUI-based PBD systems, data descriptions are often queries that can identify the correct target GUI element on which to perform the actions from all of the screen contents [163]. In our case, SUGILITE uses graphs to represent the GUI screen contents, and represents data descriptions as graph queries. Such graphs are called *UI snapshot graphs* in SUGILITE. A main challenge for PINALITE is to identify and anonymize personal information in the UI Snapshot graphs. This section summarizes the characteristics and design rationale of UI Snapshot graphs. More details can be found in Chapter 4.

Figure 8.2 shows a simple example GUI, its corresponding UI snapshot graph, and a few different example queries for describing the same button in the GUI. Each graph consists of a collection of edges represented as triples denoted as (s, p, o). The subject s and the object o are two entities, and the predicate p is a directed edge from s to o representing a relation. Each GUI element (known as a *view* in Android) is represented as an entity in the graph. They are connected to entities that represent the properties of the GUI elements, such as class types (e.g., button, check-

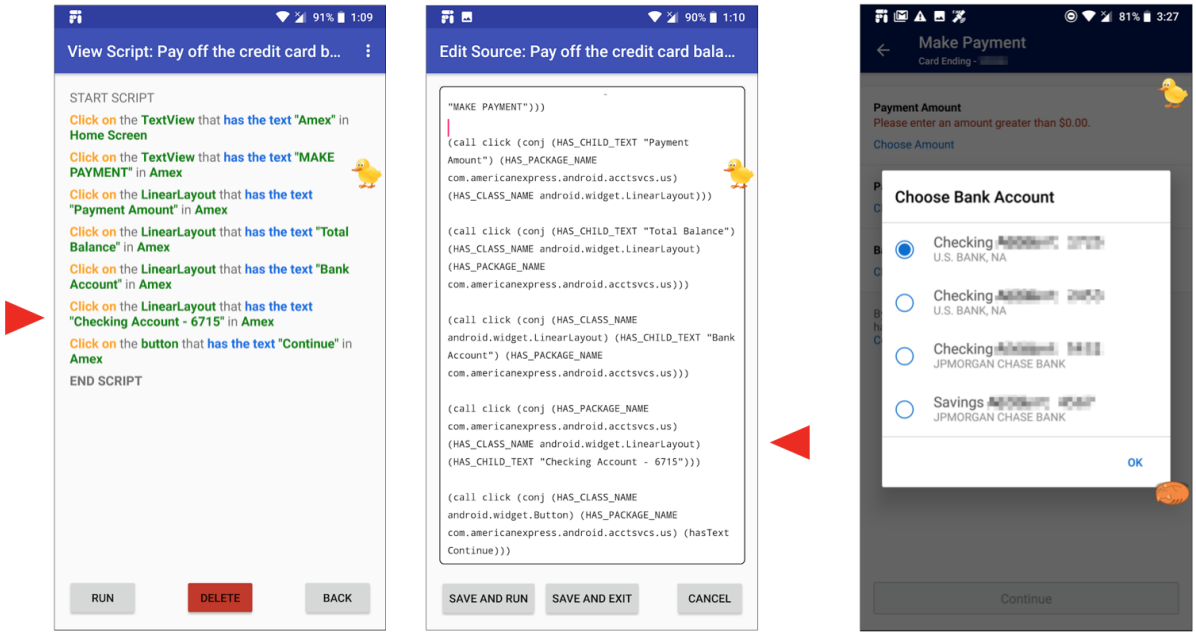


Figure 8.3: An example SUGILITE script for paying off the credit card balance using the American Express app. The screenshot (a) shows the user-readable descriptions for the operations, and the screenshot (b) shows the raw source of the script. The red triangles point to data description queries that include the user's bank account information. The screenshot (c) shows the corresponding app screen for the data description query.

box, textbox—called HAS_CLASS_NAME), on-screen bounding box coordinates (HAS_SCREEN_LOCATION), text labels (HAS_TEXT), whether the elements are clickable, scrollable or focused (IS_CLICKABLE, IS_SCROLLABLE, and IS_FOCUSED), and app-developer-assigned view IDs (HAS_VIEW_ID). The entities for GUI elements are also connected to each other based on their hierarchical relationship (i.e., HAS_PARENT and HAS_CHILD) and spatial relationship (e.g., ABOVE, BELOW, LEFT) in the current GUI layout. SUGILITE also extracts some basic semantic relations for String entities in the graph that have easily identifiable structures (e.g., CONTAINS_PRICE, CONTAINS_DATE). SUGILITE's UI snapshot graphs are generated from the hierarchical UI trees provided by Android's Accessibility API, but this approach should also apply to other types of hierarchical mobile, desktop, or web GUI representations (e.g., the DOM tree in HTML).

The UI snapshot graphs at the time of demonstrations are saved and stored with their corresponding operations and data description queries in SUGILITE scripts for a few reasons: first, they are needed to support the parameterization of scripts. In SUGILITE, parameters and their possible values are identified using a combination of the user's natural language utterance and the app GUIs during the demonstration. For example, suppose the user selects an option with text "venti" from a menu during a demonstration for a task with a spoken command "order a venti cappuccino." SUGILITE would identify "venti" as a parameter in the utterance, link it to the corresponding GUI action in the demonstration, and extract other items in the same menu (e.g., "grande" and "tall") as alternative possible values for this parameter from the UI snapshot graph.

Second, UI snapshot graphs are used to help with modifying and extending existing scripts. When the user needs to, for example, switch a data description query in use (e.g., Figure 8.2 shows three different data description queries that can be used to refer to the same UI element, the user may want to switch from one to another in an existing script), the original UI snapshot graph can be used to generate possible alternative data description queries, and validate the new data description query to ensure that it matches the user's intended target UI element for the action (described in detail in Chapter 4).

Third, UI snapshot graphs are also used for error recovery in case of execution errors. During the automation execution, when the system cannot find the target UI element for the next operation using the original data description query, the system will verify if the phone is on the correct screen of the app by comparing the current UI snapshot with the saved UI snapshot from the original demonstration. If the phone is at a different screen, it indicates a previously unseen new app state (or the app's UI has changed), so the system will prompt the user to demonstrate what to do in this new situation. For example, the Uber app will show a confirmation screen during price

surges. If this new screen was not seen during the demonstration (as determined by comparing the UI snapshot graphs), the user can show how to handle this new situation by demonstration.

8.3.3 Sources of Personal Information Leaks

This section explains possible sources of potential information leaks in shared SUGILITE scripts and their embedded UI snapshot graphs using examples.

Data Description Queries

`HAS_TEXT` and `HAS_CONTENT_DESCRIPTION` queries used in data descriptions may contain personal information of the script author. `HAS_TEXT` is used when the target UI element for an operation is identified using its displayed text string (the `text` field), and `HAS_CONTENT_DESCRIPTION` is used when the target UI element is identified using its `contentDescription` field, which is often used as the accessibility alt-text label for graphical UI elements. In many app GUIs, the contents for these fields are dynamically generated with the script author's personal information (e.g., account number, name, address—Figure 8.1 shows a few examples). Therefore, when data description queries containing them used in the operations are subsequently shared with other users, the personal information is leaked. Figure 8.3 shows an example data description query that leaks a script author's personal information. The data description queries include the last four digits of the user's bank account number, therefore the system needs to obfuscate personal information used in data description queries prior to sharing the script.

Parameter Values

Another possible source of personal information leaks is the list of possible values for parameters. As explained in Section 8.3.2, SUGILITE infers the possible values for parameters from app GUIs when the script author chooses an item from a menu by extracting the alternative items that the script author could have chosen from the same menu. Therefore, personal information can be leaked when these items are personal to the script author. For example, when a script author records the "pay off the credit card balance" script (shown in Figure 8.3), an operation can be choosing the bank account. The PBD system can infer "bank account" as a parameter, and include all the stored accounts, which are private to the script author, as possible values for this parameter in the script. To prepare the script for sharing, the system needs to obfuscate such personal information from all parameters' list of possible values.

UI Snapshot Graph Texts

As explained in Section 8.3.2, the UI snapshot graph for an app screen contains all the text strings visible on the screen. Even when the corresponding UI elements are not operated on, the text strings can still get included in the shared script to support future script modification and error handling. Although the UI snapshot graph is usually hidden from the script consumer, we do not want there to be the possibility that a script consumer could extract it. Therefore, the system also needs to examine the stored UI snapshot for each operation and obfuscate all potential personal information before sharing a script.

8.3.4 Threat Model

We assume the authors of shared scripts are anonymized, and the main goal of the attackers is to obtain enough personal information about the script authors to identify them. We assume the developers of the underlying apps used for demonstrations are generally honest and well-intentioned. The local storage of PINALITE, the client-server communication, and the server storage are all encrypted using state-of-art techniques. The attackers are able to access all the shared scripts, as well as upload a small number of scripts each with a small number of text fields to the server for the server-computed hash values before the server-side anomaly detection mechanism blocks them (see discussions in Sections 8.4.4 and 8.6).

8.4 Our Approach

In order to preserve user privacy in sharing end-user-developed scripts in GUI-based PBD systems, we present a new mechanism that can identify and mask potential personal and private information. To achieve this, we identified the following design goals:

1. The approach should require minimal user effort and intervention. In particular, it should not require script authors to manually tag personal information from a large amount of collected data.

2. The approach should retain the transparency, readability, robustness, and generalizability of the original scripts wherever possible.

3. The approach needs to detect and protect a wide range of common and uncommon, explicit and implicit types of personal information. Some of personal information may be previously unknown to the system, and dynamically generated by third-party apps.

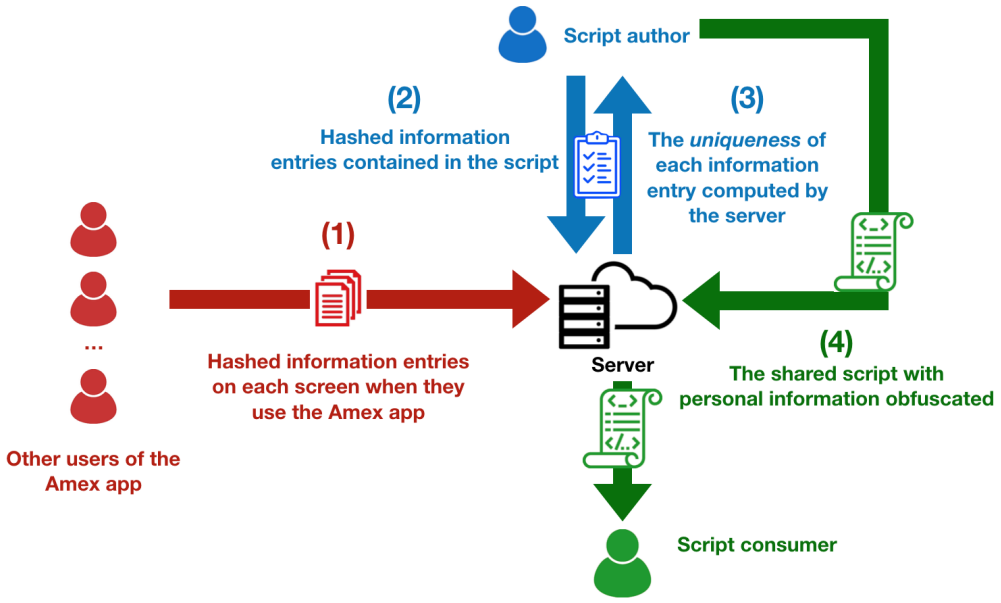


Figure 8.4: The diagram shows the pipeline of how our approach collects and aggregates information entries in an example third party app, uses those to compute the uniqueness for information entries before sharing a script, and obfuscates potential personal information in the shared script.

4. The approach should work with PBD scripts on general task domains on existing non-modified third-party app GUIs without the system having prior knowledge about the task domain or the underlying apps.

5. To ensure data security, sensitive personal information should not leave the user's device (that is, the private data should stay on the phone and not be sent to the server storing scripts or, even worse, to another person's phone).

8.4.1 Overview

Figure 8.4 illustrates the pipeline of our approach. On a high level, PINALITE identifies potential information in a PBD script based on the *uniqueness* of the information among app users. Intuitively, if a piece of information has been seen by most users, it is likely to be public, otherwise, it is possibly personal. In GUI-based PBD systems, we consider an "information entry" to be a combination of a content (i.e., a string), and its app context (i.e., the context in the app where this string appeared). Section 8.4.2 will discuss more why we define an information entry this way, and how we extract them from apps.

To determine the uniqueness of information entries, PINALITE collects and aggregates information entries in third-party mobile apps from many users in the background (Step 1 in Figure 8.4).

Potential personal information on GUIs should be different for different users (e.g., everyone should have different balance, account numbers, addresses, etc. in the Amex app), but other information should stay the same (e.g., the ”next” button and the ”contact us” phone number in an app). To protect user privacy, PINALITE only collects the *hashes* of information entries, which are sufficient for determining their uniqueness. The details of this process and the definition of “uniqueness” used in our approach are described in Section 8.4.4.

Before sharing a script, PINALITE uploads the hashes of all information entries in the script (Step 2 in Figure 8.4). The server computes and returns the uniqueness of all these information entries (Step 3 in Figure 8.4). The information entries that are more unique are considered likely personal, and therefore obfuscated using hashes by default. But the script author can also manually label and unlabeled personal information through an interactive interface. Once the author confirms that all personal information is correctly obfuscated, they can go ahead and share the script (step 4 in Figure 8.4).

After a different user, who we will call the script *consumer*, downloads a script to their personal device, PINALITE supports rebuilding obfuscated fields in the script locally using the GUI contents from the corresponding target apps on the script consumer's phone. This improves the transparency, readability, and robustness of shared scripts, and allows the script consumer to take advantage of the existing parameterization in the script. Section 8.4.6 will discuss this topic in more detail.

8.4.2 Information Entries as App Context-Content Pairs

In GUI-based PBD systems, the personal information in the scripts comes from the GUIs of the target apps. Potential personal information displayed on app GUIs can get included in the data description queries, the possible parameter values, and the UI snapshot graphs during the demonstration, as discussed in Section 8.3.3. While some kinds of GUI-displayed information are explicitly private on their own (e.g., user phone numbers), others are only private when accompanied by the contexts of app GUIs. For example, the string “New York” showing up on the checkout screen of the Starbucks app can be personal and private information since it gives away the user's location, but the same string showing up on the home screen of NYC Transit app is not personal because the app would display the string “New York” for all of its users. Even a phone number might not be private information when it belongs to a business. For example, every user will see the same phone number when visiting the “Contact Us” page in the Amex app.

Therefore, in PINALITE, we consider an information entry as a combination of a content and its app context, and use the uniqueness of such information entries to determine whether they are likely personal information.

In the implementation, the collected app context contains the unique identifier of an app: *package name* and the unique identifier of a screen within an app: *activity name*. During the background data collection and the recording of a script, PINALITE's client-side software collects all text strings visible on the screen along with their app contexts as *app context-content pairs*. That is, an app context-content pair denotes "the information X is displayed on the screen Y in the app Z."

8.4.3 Background Data Collection

After PINALITE's client-side software is installed on a user's phone, and given the appropriate permissions, it begins collecting information in the background, which it sends to the server (Step 1 in Figure 8.4). On each user's phone, when not recording, the client-side software collects app context-content pairs (defined in Section 8.4.2) in the background for the current app running on the phone, and uploads the hash values of them to the server (the hashes are computed locally). Our implementation uses the SHA-512 hash algorithm for all our hash functions. It is a one-way hash function, which means that is practically infeasible to derive the original string from the hash value. The uploaded data are labeled with the user's Android *Advertising ID* [96], which is an anonymous, user-specific, unique, and resettable user ID. The server keeps a count of the number of unique users with identical information entries (i.e., how many users have seen the same information on the same screen of an app) and the number of unique users with the same app context (i.e., how many users have visited the same screen of an app).

To preserve user privacy, the user only sends out the hashes of the app context and the contents in information entries, so neither any app content nor the app usage can be recovered from the uploaded data. For an information entry, the user only uploads $hash(\text{app\_context}, \text{content})$ to the server, so that the plain values of private information do not leave the user device. When receiving the user-uploaded hashes of information entries, the server applies another level of server-side hash with an encrypted salt before storing them (i.e., the data stored on the server for an information entry is $hash'(hash(\text{app\_context}, \text{content}), \text{salt})$. This prevents dictionary attacks [230] (i.e., the attacker enumerates the hashes of all possible content values—for example, if the attacker guesses that one of the hash values in an uploaded app UI snapshot may represent the user's city, they can enumerate the hashes for all possible city names, and compare if any matches a hash in the uploaded app UI snapshot) and rainbow table attacks [220] (similar to the dictionary attack, but the attacker looks up a pre-computed table of hash values for strings up to a certain length consisting

of a certain set of characters), because even the attackers got access to the hashes, they would not be able to decrypt the data because they cannot compute hashes offline without the encrypted salt.

8.4.4 Identifying Personal Information for PBD Script Sharing

The next step is to identify personal information in scripts. PINALITE uses the *uniqueness* of an information entry among all users of the same to determine if it is likely to be personal. As described in Section 8.4.3, the server collects and aggregates hashed information entries from app GUIs of many users. Information entries that only appear on one or a very small number of users' devices are likely personal, while entries that appear across devices of many users are likely not. An advantage of this approach is that it makes re-identification attacks difficult. It not only protects common types of explicit personal information, but also identifies and masks any information entries that are non-common among users. As a result, information entries that are classified as non-personal are those shared by the majority of users. Therefore it would be very difficult to re-identify a user using these information entries.

Before sharing a script, PINALITE's client-side software for the script author first queries for the uniqueness of each information entry in the script (Step 2 in Figure 8.4). For each entry, it sends out $hash(\text{app\_context}, \text{content})$ in the query to the server. The server then computes the salted hash value $hash'(hash(\text{app\_context}, \text{content}), \text{salt})$ using the encrypted salt, and counts the number of unique users with the identical information entries (i.e., the number of users who have seen this information on this screen in this app) and the number of unique users who have had the same app context (i.e., the number of users who have been to this screen in this app).

With these counts, the server computes the uniqueness of each information entry by estimating the probability of a user has seen the same content given the user has been in the same app context. The details of the process are described below:

Let's define the observed number of unique users $F(E,U,a,c)$ as the number of users in a dataset of information entries E who have had the identical app context-content pair (a,c) from the user set U of all the users that have contributed to E:

$$F(E,U,a,c) = \sum_{u \in U} [\exists e \in E : (\text{user}_e = u) \wedge (\text{app\_context}_\text{e} = a) \wedge (\text{content}_\text{e} = c)]$$

Similarly, we define the observed number of unique users $G(E,U,a)$ as the number of users in a dataset of information entries E who have had the same app context a from the user set U of all the users that have contributed to E:

$$G(E,U,a) = \sum_{u \in U} [\exists e \in E : (\text{user}_e = u) \wedge (\text{app\_context}_e = a)]$$

Therefore, for an app context-content pair (a,c), the server can calculate the observed frequency $H(E,U,a,c)$ of the users who had the app context-content pair (a,c) in all the users who have had the app context a:

$$H(E,U,a,c) = \frac{F(E,U,a,c)}{G(E,U,a)}$$

The result of $H(E,U,a,c)$ estimates the probability that a user has seen the content c given that the user has been in the app context a. The server uses this estimated probability to indicate the uniqueness of the content c in the app context a.

Lastly, the server tests if this probability meets a pre-specified threshold t using the one-tailed exact test of goodness-of-fit [196]. The null hypothesis is that (a,c) does not meet the uniqueness threshold t—the probability of a user has seen the content c when the user has been in the app context a is greater than t, and the alternative hypothesis is that the probability of a user has seen the content c when the user has been in the app context a is less than or equal to t.

If the result indicates that the probability is less than or equal to t with statistical significance[2], we consider the app context-content pair (a,c) to potentially contain user personal information that needs to be obfuscated. Specifying a smaller t lowers the risk of false negatives (i.e., classifying personal information as non-personal) in the shared script, especially when the sample population is skewed (see Section 8.6 for the detailed discussion). However, it increases the likelihood of false positives (i.e., classifying non-personal information as personal), and specifically, requires a larger number of users to collect UI snapshots from before non-personal information can be confidently identified.

For example, at $t = 0.5$ (which means that for an information entry to be non-personal, at least a half of users who have been to this screen of this app should have seen the same information), our approach requires collected UI snapshots from 5 unique users (with the threshold of significance $p < 0.05$) that *all* have seen the same information before it can label this information entry as non-public. Before that, it would classify everything as potentially personal.

8.4.5 The Interactive Interface

When the script author chooses to share a script, PINALITE first identifies personal information in the script using the method described above in Section 8.4.4, and then displays the result in an

[2]We used $p < 0.05$ in our implementation.

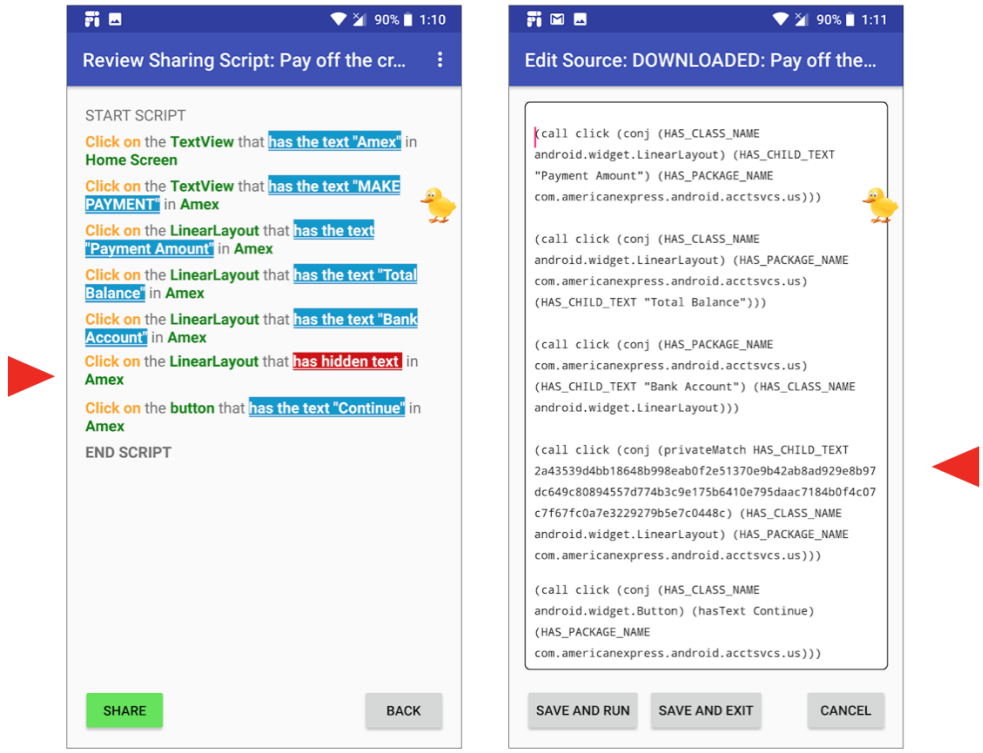


Figure 8.5: The result of processing the script shown in Figure 8.3. The left side shows the review interface for the script author, where the identified personal information is highlighted in red. The right side shows the source of the processed script, where the content of identified personal information is replaced by its hash.

interactive interface (shown in Figure 8.5a). For information entries that have been classified as public, the interface displays them in plain text, and highlights them in blue. Those that have been classified as potentially personal are marked as "hidden text" and highlighted in red. The script author can review the script to be uploaded, and manually label or unlabel private information as needed by clicking on them.

If an information entry is labeled as public, it will be included in the shared script in plain text, otherwise, the shared script will include its encrypted salted hash ($hash'(hash(\text{app\_context}, \text{content}), \text{salt})$). Server-computed salted hashes are used for obfuscating the shared scripts in order to prevent dictionary attacks and rainbow table attacks, as discussed in Section 8.4.3. Once the script author confirms the script, PINALITE will upload the obfuscated script to the server for sharing.

8.4.6 Rebuilding Scripts with Obfuscated Information by Script Consumers

For SUGILITE scripts, obfuscating string values in data description queries or parameter values by hashing does **not** necessarily prevent executing scripts with hash values in data descriptions. Data description queries in SUGILITE compare the *equality* of string references in the query (e.g., click on the button that has text "next") with text labels of GUI elements on the screen. With hashed values, the comparison can still be done on the server-side, because the hash values in the scripts are originally computed by the server using the encrypted salt. PINALITE's client-side software can send the hashed UI snapshot to the server during the script execution, from which the server can return which, if any, of the elements on the current UI snapshot matches the hashed query.

However, in practice, the script execution using only equality checks often fails for the consumers of shared scripts when the data description queries contain the hash values of string contents that are personal to the original script author. When SUGILITE executes an action, it will not be able to find the target UI element because the string content used in the data description query can match anything in the app on the script consumer's phone. For example, the script in Figure 8.3 will not work with a different user, because the new user presumably does not have an account with the same account number. Therefore the system needs to rebuild the obfuscated data descriptions using the script user's local information.

There are some additional motivations for rebuilding obfuscated data descriptions: first, it helps improve script transparency. It allows script consumers to understand script behaviors, to check to see if the script fits the script consumer's needs, and allows the script consumer to modify and extend the scripts when needed. Second, as discussed earlier, comparing salted hash values can only be done on the server-side. Rebuilding the obfuscated fields in scripts allows the script

consumer to execute them on the consumer's phone without help from the server. Lastly, when the script parameter values are personal to the script author, rebuilding them on the script consumer's side allows the script consumers to take advantage of the parameterization with their own personal options. For example, a script consumer can replace the data description query in the highlight operation in the screen shown in Figure 8.3 with their own account numbers, and use them as possible parameter values for the parameter "account number" so that this script can be used for all of the script consumer's bank accounts.

Rebuilding Obfuscated Fields

Here we explain how PINALITE rebuilds the obfuscated fields in shared scripts locally on the consumer's phone. In the earlier example shown in Figure 8.3, when a script author records a script that pays off the credit card balance using the Amex app, one step is to choose an account to pay from. This action is recorded as "Click on the LinearLayout that has the text 'Checking Account ...1234 in Amex". Since the author chooses this account from a menu that also contains options for the author's other accounts, the PBD system can recognize "account" as a parameter, and save the other account options as possible values for this parameter so that this script can later be used for paying off the credit card balance using the other accounts (see more details about the script parameterization process in Chapter 3). However, when PINALITE prepares the script for sharing, both the original data description text ("Checking Account ...1234") and the alternative options will be obfuscated because these information entries are all personal to the script author. When script consumers download this script, all they can see is "Click on the LinearLayout that *has hidden text* in Amex" (shown in Figure 8.5a). Under the hood, the hidden text represents a salted hash of the original text in the source, as shown in Figure 8.5b. This script will not execute correctly, because the script consumer would not have an account whose text label matches the hash of the original account text label of the script author. The parameter mechanism would be broken too for the same reason.

A prerequisite to our solution is to add a step in the script recording mechanism on the script author's side. Each time the script author demonstrates an operation, in addition to generating the main data description that best reflects the script author's intention, the recording mechanism *also* extracts an alternative data description query that uses no potential personal information on app GUIs. As explained in Section 8.3.2, each app screen is represented as a GUI snapshot graph. Therefore, the alternative data description needs to be a graph query that (1) uniquely identifies the target UI element in the graph, and (2) does not use the (`HAS_TEXT`) relation on strings containing any potential personal information. For the credit card payment example used above, an alterna-

tive to the original data description "Click on the TextView that has the text 'Checking Account (...1234) in Amex" can be "Click on the first TextView in the ListView below the TextView that has the text "Choose Bank account" in Amex." In this alternative query, the text "Choose Bank account" is not personal, because everyone who has visited the account selection screen in the Amex app can see the same text on the screen.

On the script consumer's phone, PINALITE rebuilds the script at the runtime using the script consumer's local app GUI information. When executing a script, when SUGILITE reaches the account selection menu, it cannot find any screen element with text that matches the hash value in the script. At this time, the system uses the previously generated alternative data description query to locate the account selection menu on the screen, replaces the "hidden text" data description with the new value (e.g., "Checking Account (...2345)" found locally on the consumer's phone from the matched UI element, and extracts other menu options from the GUI (e.g., "Saving Account (...3456)") to be used as the new possible parameter values *for this specific script consumer*. This allows the script consumer to see the data description queries revealed in the script (e.g., the script consumer's own account information for the script in Figure 8.5 instead of "hidden text"), and to take advantage of the parameterization in the script using their own personal information.

An underlying assumption of this approach is that the SUGILITE operates on the script consumer's personal device, and the script consumer does not share the device with others. Our approach assumes that the script consumer *should* have access to any information the system sees in app GUIs locally on their phone, and therefore the information can be safely used to rebuild the script consumer's personal copy of the automation script. Similarly, other people should not have access to the script consumer's personal copy of the script stored locally on the phone, since it may contain the script consumer's personal information.

8.5 Evaluation

We evaluated our approach on three research questions:

RQ1 How accurate is PINALITE's approach in identifying personal information in GUI-based PBD scripts?

RQ2 How well can script consumers understand the shared scripts after they are obfuscated using PINALITE?

RQ3 How comfortable do script authors feel about sharing their scripts with PINALITE?

We investigated the first question through an accuracy evaluation, and the last two questions through a lab user study.

8.5.1 Accuracy of Identifying Potential Personal Information

Method

We selected 15 popular Android apps from various task domains in the Google Play Store. For each app, we came up a task that (1) reflects the main purpose of the app, (2) is reasonable to be automated using an agent, and (3) interacts with some personal information of the user. We performed each task with 5 different "simulated users", and collected the corresponding information entries from these UI snapshots. For each simulated user, We created a new app account using different a profile, and spoofed a new GPS location. (Note that we did not select, for example, banking apps and medical apps in this evaluation because it was difficult to create new accounts for simulated users in these apps.) We then created a SUGILITE script for each target task by demonstration, and used PINALITE to identify personal information in the script with the uniqueness threshold $t = 0.5$.

For each script, we extracted all of the information entries contained in its data description queries, parameter values, and UI snapshots. We evaluated the performance of our approach on these scripts in terms of *precision* (the fraction of information entries that contain personal information among the ones classified as "potentially personal"), *recall* (the fraction of information entries that contain personal information that were correctly classified as "potentially personal"), and *accuracy* (the fraction of information entries that were correctly classified) of identifying personal information by comparing the result against expert human judgments. We used two experts to independently label whether each information entry contains any personal information. The two judges reached *substantial agreement* [144], with the percent of agreement = 87.8% and Cohen′s $\kappa = 0.63$.

Results and Discussion

Table 8.1 reports the result of the evaluation. First of all, note the large number of strings in each script (column n), ranging from 42 to 193 strings—too many for script authors to comfortably review manually. All 15 scripts included some personal information of the script author (column n_{personal}). Our approach achieved a perfect recall—that is, all personal information entries were indeed correctly identified as potentially personal, and therefore obfuscated. No personal information was leaked in the processed scripts in our evaluation. This result suggests that our approach is adequate in preserving the script author's personal information in end-user developed PBD scripts.

App	Task	n	n_{personal}	Recall	Prec.	Accu.
Starbucks	Order a cup of coffee for pick up	58	11	1	0.38	0.83
Uber	Request a ride	128	35	1	0.74	0.69
Marriott	Book a hotel room	140	8	1	0.44	0.93
Papa John's	Order a pizza for delivery	142	5	1	0.19	0.85
AccuWeather	Check the current weather	59	3	1	0.17	0.75
OpenTable	Make a restaurant reservation	116	12	1	0.26	0.71
Kayak	Book a flight	89	21	1	0.51	0.78
AMC Theatres	Buy a movie ticket	91	6	1	0.12	0.53
Zillow	Check the homes for sale in the area	42	4	1	0.25	0.71
Lyft	Request a ride	68	37	1	0.86	0.85
Cinemark Theatres	Buy a movie ticket	193	9	1	0.11	0.60
Chipotle	Order food for pickup	136	34	1	0.89	0.94
Yahoo! Sports	Check sports scores	86	13	1	0.34	0.71
Dunkin'	Order a drink for pick up	76	19	1	0.83	0.95
Uber Eats	Order food for delivery	63	13	1	0.41	0.73

Table 8.1: The result of the evaluation, showing the app name, the task, the number of information entries collected (n), the number of information entries that contained personal information (n_{personal}), the recall, the precision, and the accuracy for each script.

On the other hand, the precision of our approach was lower in some task domains. That is, our approach sometimes classifies non-personal information as personal information. This trade-off was a deliberate design decision we made—our approach seeks to minimize false negatives by not limiting to pre-specified common types of personal information such as phone numbers, email addresses, and birthdays. Instead, it by default obfuscates all "unique" information, defined as information that is not seen by most people who have visited the same screen in the same app. This approach ensures a higher degree of privacy for users. A trade-off of this approach is that it may introduce false positives by obfuscating information that people might not consider private. When we looked into the false positives in our evaluation, they were mostly (1) personalized recommendations and advertisements—fields that show different contents for different users such as movie recommendations in AMC and Cinemark apps; and (2) dynamic information that depends on the time and location, such as the temperature readings in AccuWeather, the available timeslots for restaurants in OpenTable, the date selections in Marriott, and the latest sports news in Yahoo!

Sports. It is also worth noting that these kinds of false positives are unlikely to impact the function and the usability of the shared PBD scripts. These fields are almost never used in data descriptions for script actions, and are rarely useful for script generalization and error handling because they are not consistent across multiple instances of script execution.

Our approach successfully identified many kinds of uncommon and seemingly innocent personal information that may be combined for re-identification attacks, such as the address of and the distance to nearby locations in Starbucks, Papa John's, and AMC Theatres, the addresses of nearby real estate properties in Zillow, the amount of reward points in Starbucks and Marriott, the names of local sports teams in Yahoo! Sports, and the past order history of the script author in Uber Eats. This was in addition to the common types of explicit personal information identified in the scripts such as the script author's name in greetings ("Good morning, Bob"), the stored script author's home address, and the script author's account name. When labeling the ground truth for personal information, we did **not** count information entries that may only reveal the script author's personal information with some prior domain knowledge, such as a product offering that is only available to users in a certain region, and a menu option that only shows up for users with certain reward status. In our metric, those were counted as false positives, but they were protected using our approach nevertheless.

8.5.2 User Study

Participants

We recruited 16 participants (5 identified as female, 11 identified as male, ages 21–32) for our user study. The user study session for each user lasted around 30-40 minutes. We compensated the participants $15 for their time. Most (75%) participants were students at local universities. The others worked different technical, administrative, or managerial jobs. Among the participants, there were 4 (25%) non-programmers (i.e., those who have done none or only very light programming such as writing Excel functions), 5 (31.3%) novice programmers (i.e., those who have taken 1-2 college-level computer science classes, or with equivalent expertise, but have little "real-world" programming experience), and 7 (44%) more experienced programmers. 5 (31.3%) participants had some prior experiences with PBD systems (e.g., macro recorders in MS Office and Adobe software), while the other 11 participants (68.7%) had no prior experience with PBD.

Methods

For each study session, after obtaining the proper consent and having the participant fill out a demographic survey, we gave the participant a short tutorial on the general concept of GUI-based PBD, and how SUGILITE works using its published tutorial video[3]. After the tutorial, we investigated RQ2: How well can script consumers understand the obfuscated scripts?

We gave each participant 3 tasks in random order. The goal of the tasks was to simulate the experience of understanding a shared script for a script consumer. For each task, the participants saw an obfuscated script like in Figure 8.5a. We used the Amex example shown in Figure 8.5, and the Uber and Dunkin' scripts that were also used in the accuracy evaluation (Table 8.1). The participants saw screens of running the script on the phone of a simulated user different from the script author (so that the screens would show different contents in the personal information fields). The user experience in this process should be identical to that when a script user downloads and executes a shared script using PINALITE. We used a simulated user profile instead of having the participants create new user profiles in the study in order to avoid accidentally collecting any actual personal information from the participants. We then asked the participant to tell us what the purpose of the script was, and what the "hidden text" represented in the script. Finally, each participant rated if it was easy for them to understand what each script does.

Following the tasks, we moved to answering RQ3 (How comfortable do script authors feel about sharing their scripts with PINALITE?) using a questionnaire. We first described the concept of uniqueness in our approach, explained the guarantee that our approach provides (for an information entry to be non-personal, at least a half users who have been to this screen of this app should have seen the same information), and showed the review interface using the Amex script example. We did *not* have the participants demonstrate new scripts because (1) we wanted to exclude the factor of the usability of SUGILITE's recording mechanism (which has been evaluated in Chapter 3) from this study; and (2) we wanted to avoid accidentally collecting any actual personal information from the participants. Each participant then filled out a questionnaire on how much they trusted the system as a script author, how comfortable they were with sharing scripts in our system, and whether the review interface was clear about what information is hidden and what information is revealed in shared scripts. Finally, we ended the session with an informal interview on their thoughts about our approach. We specifically asked if they had any privacy concerns with our approach, and whether they could think of any scenarios where they would not be comfortable sharing scripts processed with our approach.

[3] `https://www.youtube.com/watch?v=KMx7Ea6W6AQ`

Results and Discussion

The results for RQ2 are encouraging. Our participants found the obfuscated scripts easy to understand. After completing the 3 tasks of understanding obfuscated shared scripts, they on average rated 4.69 out of 5 (standard error of the mean $\sigma_{\bar{x}} = \pm 0.12$) on a 5-point Likert scale for the statement "*I find it easy to understand what each script does*", and on average rated 4.44 out of 5 ($\sigma_{\bar{x}} = \pm 0.19$) for the statement "*The script representation is useful for me to determine if the script serves my needs*" in the questionnaire.

In the interview, the participants confirmed our hypothesis that script consumers would like to review the scripts from others before executing them. For example, P9 agreed they "were concerned about using others' scripts, [and they wanted to check] whether that contains the virus or other unwanted hidden procedures." However, several participants, although finding the current representation clear and easy-to-understand, also thought it to be too difficult to go through the entire script. P15 said, "*...there must be some easier way to do it.*" P3 and P12 also worried that some words used in the script representation (e.g., "TextView" and "LinearLayout") were too technical for end users (both P3 and P12 were experienced programmers). These findings identified the research opportunity of designing an easier-to-use script representation of PBD scripts specifically for script consumers. But overall, participants did not have much of a problem with the transparency of the scripts with obfuscated fields. They found it easy to understand the purpose of each script and what each "hidden text" might represent as a script consumer, with the help of referencing the GUIs of the underlying third-party apps.

The results for RQ3 are also generally positive. 14 (87.5%) participants reported that they felt comfortable sharing their personal scripts with others using our approach in the study. 14 (87.5%) participants found the script review interface (shown in Figure 8.5a) clear about what information was private and what information was public in the scripts. However, there were 2 participants who reported that they could not trust PINALITE to protect their personal information.

Some of the distrust originated from the sensitivity of certain task domains. For example, in the interview, P1 reported, "*I might not let a virtual agent do the task related to my bank account.*" but was comfortable sharing scripts in other task domains with PINALITE. Another participant who reported distrust in our system, P6, said, "*It is not about this mechanism or app, I just don't trust technology in general*," and later added, "*but sharing with friends is acceptable.*" A few other participants, while comfortable with sharing scripts with the public using PINALITE, also mentioned that scripts shared only with friends can have different standards for privacy. P4 and P7 wanted to include more personalizations in their scripts to make them more effective for their friends to use. P4 said, "*I want to share with my friends about ...(some customization of coffee)*

since I think my recipe is delicious. But that information would be definitely hidden with your mechanism." Lastly, a few participants reported non-privacy-related concerns about script sharing. P4 was not sure if his scripts are effective enough for other people to use, and P13 said "*... I don't want to be responsible for the performance of my scripts.*"

8.6 Limitations

Our current approach has some limitations. First, the current version of PINALITE only processes textual information. It can not identify and obfuscate personal information from visual information in images, or other multimedia forms. This limitation does not pose a problem when working with the current version of the underlying SUGILITE PBD system. SUGILITE, like many other GUI-based PBD systems such as CoScripter [152], PLOW [3], and ActionShot [155] do not use information about the contents of images and other multimedia forms in scripts. So they do not collect any visual information in their scripts. But for our approach to work with vision-based PBD systems (e.g., Sikuli [299] and HILC [120]), it would need to be able to identify and obfuscate visual information as well.

Second, PINALITE also does not consider situations where the layout of an app encodes personal information. For example, suppose a layout structure in an app only shows up for users in a specific user group (e.g., the entire "loan" menu in a banking app only shows up for users with outstanding loans), then the presence of references to this GUI layout structure in the alternative queries used for rebuilding scripts may reveal personal information about the script author.

Third, when testing whether the uniqueness of an information entry meets the threshold t, we assume that the population of sample users is independent. This assumption can be a problem when the sample population is *very significantly* skewed. Let's look back at the example in Section 8.4.2, the string "New York" on the checkout screen of the Starbucks app should be identified as potential personal information, as it gives away the script author's current city. However, if the system encounters a skewed sample population where the vast majority of users are actually from New York, the system may classify it as a non-personal information entry, and therefore reveal it by default in future shared scripts. One way to mitigate this problem is to impose more strict thresholds for t and for the p value used in the statistical test. But this comes with the trade-off of having more false negatives (non-personal information classified as personal information). It is worth noting that this limitation only affects "less differentiating" personal information such as the current city, gender, and country when the sample population is significantly skewed. More sen-

sitive information such as names, exact addresses, account numbers will not be impacted because these are more likely to be unique among the users.

Fourth, PINALITE uses a server-side salted hash approach (see Section 8.4.4) to prevent dictionary attacks. The attackers cannot decrypt obfuscated personal information by comparing the hashes appeared in scripts against offline-computed hashes for a large number of possible values because computing hashes requires access to the encrypted salt. The server can control the access to the salted hash function and block abnormal requests (e.g., those with abnormally large numbers of strings, or from clients with too many requests). However, for fields with only a few possible values (e.g., gender), it is still possible for an attacker to enumerate all possible values in the server requests without being detected and blocked. But this problem will not affect more sensitive personal information fields like names, exact addresses, and account numbers, which have larger numbers of possible values. In our threat model (Section 8.3.4), we assume that the authors of shared scripts are anonymized, and the main goal of the attackers is to obtain enough personal information about the script authors to identify them. Therefore, fields with only a few possible values are not very useful (e.g., knowing a script author is a female Starbucks app user between 25-39 is not sufficient for identifying her). Since shared scripts are anonymized, and hashes are computed with app contexts (i.e., the same string will result in different hash values when appearing in different apps, or different screens within the same app), the attackers are not able to link information across multiple scripts to further deanonymize the script author either. However, when the identity of the script author is known, such as when a script author directly shares a PINALITE-obfuscated script with others, potential sensitive information with a small number of possible values (e.g., gender and age group) may get compromised when the attackers try enumerating the possible options with the PINALITE server to obtain their hashes.

In practice, PINALITE can also be combined with other methods to support identifying potential personal information at a finer granularity. For example, one can use the "personal database" method used in CoScripter [152] to manually blacklist specific fields known to contain personal information (e.g., gender), and use regular expressions to explicitly identify personal information with fixed structures like email addresses and phone numbers. This would allow the system to obfuscate these fields using named types like "hidden email address" and "hidden user name" instead of the generic "hidden information", further improving the readability and generalizability of shared scripts.

8.7 Future Work

The current version of PINALITE addresses the privacy-preserving sharing of individual scripts. For the future, it would be interesting to explore aggregating scripts for similar tasks from multiple users. This would allow us to collect data about task automation procedures on different phones in various environments, and create more robust scripts that can handle differences in target app versions, screen layouts, phone usage contexts, and task parameters. Prior to the introduction of an effective privacy-preserving mechanism, we cannot collect runtime data of GUI-based PBD systems which contain actual script contents in the background, since the contents may contain the script author's personal information. With PINALITE, it may be feasible to extract runtime data of GUI-based PBD systems in the background, which would allow the PBD system to collect more data for aggregation purposes.

Another interesting future research direction is to better communicate with the script authors and the script users about the data they may share and the privacy expectations associated with sharing these data. This should help users become more comfortable with participating in the background data collection and sharing their scripts, while making informed decisions on whether to share a script and whether to mask/unmask a field in a shared script at the script review stage.

8.8 Chapter Conclusion

This chapter presents a new mechanism, called PINALITE, for identifying and obfuscating possible personal information in SUGILITE scripts based on the uniqueness of information entries in app GUIs. By collecting hashed samples of GUI screens from users, PINALITE is able to determine if a piece of information is likely personal by estimating the probability that a user has seen this information when that the user has visited this screen in the app. PINALITE's approach can also replace the obfuscated fields in a shared script locally using the script consumer's app GUI contents, preserving the transparency, readability, robustness, and generalizability of the original script.

Chapter 9

The Semantic Representation of GUIs

9.1 Introduction

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 SUGILITE and others (e.g. [247]) use task-relevant entities and hierarchical structures extracted from GUIs to parameterize, disambiguate, and handle errors in user-demonstrated task automation scripts. ERICA [67] 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 [168] extracts task flows from mobile app GUIs to bootstrap conversational agents.

Semantic representations of GUI screens and components, where each screen and component is encoded as a vector (known as the *embedding*), are highly useful in these applications. The representations of GUI screens and components can be used to also represent other entities of interest. For example, a task in an app can be modeled as a sequence of GUI actions, where each action can be represented as a GUI screen, a type of interaction (e.g., click), and the component that is interacted with on the screen. An app can be modeled as a collection of all its screens, or a large collection of user interaction traces of using the app. Voice shortcuts in mobile app deeplinks [11] can be modeled as matching the user's intent expressed in natural language to the

This chapter is modified from the conference paper: Toby Jia-Jun Li, Lindsay Popowski, Tom M. Mitchell, and Brad A. Myers. Screen2Vec: Semantic Embedding of GUI Screens and GUI Components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI 2021)*.

target GUI screens. The representation of the screen that the user is viewing or has previously viewed can also be used as the context to help infer the user's intents and activities in predictive intelligent interfaces. The semantic embedding approach represents GUI screens and components in a *distributed* form [20] (i.e., an item is represented across multiple dimensions) as continuous-valued vectors, making it especially suitable for use in popular machine learning models.

However, existing approaches of representing GUI screens and components are limited. One type of approach solely focuses on capturing the text on the screen, treating the screen as a bag of words or phrases. For example, the original SUGILITE system (Chapter 3) uses exact matches of text labels on the screen to generalize the user demonstrated tasks. The SOVITE interface (Chapter 7) uses the average of individual word embedding vectors for all the text labels on the screen to represent the screen for retrieving relevant task intents. This approach can capture the semantics of the screen's textual content, but misses out the information encoded in the layout and the design pattern of the screen, and the task context encoded in the interactivity and meta-data of the screen components.

Another type of approach focuses on the visual design patterns and GUI layouts. ERICA [67] uses an unsupervised clustering method to create semantic clusters of visually similar GUI components. Liu et al.'s approach [183] leverages the hierarchical GUI structures, the class names of GUI components, and the visual classifications of icons to annotate the design semantics of GUIs. This type of approach has been shown to be able to determine the category of a GUI component (e.g., list items, tab labels, navigation buttons), the "UX concept" semantics of buttons (e.g., "back", "delete", "save", and "share"), and the overall type of flow of screens (e.g., "searching", "promoting", and "onboarding"). However, it does not capture the content in the GUIs—two structurally and visually similar screens with different content (e.g., the search results screen in a restaurant app and a hotel booking app) will yield similar results.

There have been prior approaches that combine the textual content and the visual design patterns [171,226]. However, these approaches use supervised learning with large datasets for specific task objectives. Therefore they require significant task-specific manual data labeling efforts, and their resulting models cannot be used in different downstream tasks. For example, Pasupat et al. [226] create an embedding-based model that can map the user's natural language commands to web GUI elements based on the text content, attributes, and spatial context of the GUI elements. Li et al.'s work [171] describes a model that predicts sequences of mobile GUI action sequences based on step-by-step natural language descriptions of actions. Both models are trained using large manually-annotated corpora of natural language utterances and the corresponding GUI actions.

We present a new *self-supervised* technique (i.e., the type of machine learning approach that trains a model without human-labeled data by withholding some part of the data, and tasking the network with predicting it), called `Screen2Vec`, for generating more comprehensive semantic representations of GUI screens and components using their textual content, visual design and layout patterns, and app context meta-data. `Screen2Vec`'s approach is inspired by the popular word embedding method `Word2Vec` [202], 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 [66] dataset.

Along with this chapter, we also release the open-source[1] code of `Screen2Vec` as well as a pre-computed `Screen2Vec` model trained on the RICO dataset [66] (more in Section 9.2.1). The pre-computed model can encode the GUI screens of Android apps into embedding vectors off-the-shelf. The open-source code can be used to train models for other platforms given the appropriate dataset of user interaction traces.

`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., [167, 247]), intelligent suggestive interfaces (e.g., [52]), assistive tools (e.g., [27]), and GUI design aids (e.g., [149, 268]). `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,

This chapter makes the following contributions:

1. `Screen2Vec`: a new self-supervised technique for generating more comprehensive semantic embeddings of GUI screens and components using their textual content, visual design, layout patterns, and app meta-data.

2. An open-sourced GUI embedding model trained using the `Screen2Vec` technique on the RICO [66] dataset that can be used off-the-shelf.

[1] A pre-trained model and the `Screen2Vec` source code are available at: `https://github.com/tobyli/screen2vec`

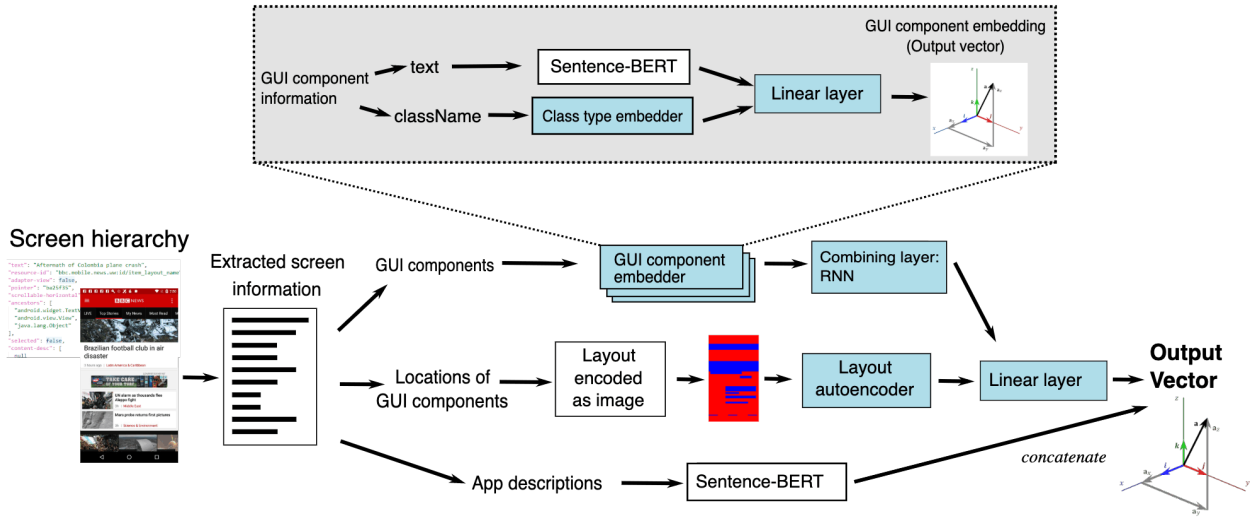


Figure 9.1: The two-level architecture of `Screen2Vec` for generating GUI component and screen embeddings. The weights for the steps in teal color are optimized during the training process.

3. Several sample downstream tasks that showcase the model's usefulness.

9.2 Our Approach

Figure 9.1 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 first describe the approach at a high-level here, and then explain the details in Section 9.2.2.

The GUI component level model encodes the textual content and the class type of a GUI component into a 768-dimensional[2] 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 [238] model; and (2) a 6-dimensional class embedding vector that represents the class type of the GUI component, which we will discuss in detail in Section 9.2.2. The two embedding vectors are combined using a linear layer (Section 9.2.2), 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 com-

[2]We decided to produce 768-dimensional vectors so that they can be directly used with the 768-dimensional vectors produced by the pre-trained Sentence-BERT model with its default settings [238].

ponents 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), which we will discuss more in Section 9.2.2; (2) a 64-dimensional layout embedding vector that encodes the screen's visual layout (details later in Section 9.2.2); and (3) a 768-dimensional embedding vector of the textual App Store description for the underlying app, encoded with a pre-trained Sentence-BERT [238] model. These GUI and layout vectors are combined using a linear layer (Section 9.2.2), 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 because the RICO dataset does not include interaction traces that go across multiple apps). 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.

9.2.1 Dataset

We trained `Screen2Vec` on the open-sourced RICO[3] dataset [66]. 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. For each GUI screen, the RICO dataset includes a screenshot image (that we did not use in `Screen2Vec`) and the screen's "view hierarchy" in a JSON file. The view hierarchy is structurally similar to a DOM tree in HTML; it starts with a root view, and contains all its descendants in a tree. The node for each view includes the class type of this GUI component, its textual content (if any), its location as the bounding box on the screen, and various other properties such as whether it is clickable, focused, or scrollable, etc. Each interaction trace is represented as a sequence of GUI screens, as well as information about

[3]Available at: `http://interactionmining.org/rico`

which (X, Y) screen location was clicked or swiped on to transit from the previous screen to the current screen.

9.2.2 Models

This section explains the implementation details of each key step in the pipeline shown in Figure 9.1.

GUI Class Type Embeddings

To represent the class types of GUI components, we trained a class embedder to encode the class types into the vector space. We used a total of 26 class categories: the 22 categories that were present in [183], one layout category, list and drawer categories, and an "Other" category. We classified the GUI component classes based on the classes of their `className` properties and, sometimes, other simple heuristic rules (see Table 9.1). For example, if a GUI component is an instance of `EditText` (i.e., its `className` property is either `EditText`, or a class that inherits from `EditText`), then it is classified as an Input. There are two exceptions: the Drawer and the List Item categories look at the `className` of the parent of the current GUI component instead of the `className` of itself. A standard PyTorch embedder (`torch.nn.Embedding`[4]) maps each of these 26 discrete categories into a continuous 6-dimensional vector. The embedding vector value for each category is optimized during the training process for the GUI component prediction tasks so that GUI components categories that are semantically similar to each other are closer together in the vector space.

GUI Component Context

As discussed earlier, `Screen2Vec` uses a Continuous Bag-of-Words (CBOW) prediction task [202] for training the weights in the model, where for each GUI component, the model tries to predict it using its context. In `Screen2Vec`, we define the context of a GUI component as its 16 nearest components. The size 16 is chosen to balance the model performance and the computational cost.

Inspired by prior work on the correlation between the semantic relatedness of entities and the spatial distance between them [169]. We tried using two different measures of screen distance for determining GUI component context in our model: `EUCLIDEAN`, which is the straight-line minimal distance on the screen (measured in pixels) between the bounding boxes of the two GUI components; and `HIERARCHICAL`, which is the distance between the two GUI components on

[4] `https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`

GUI Component	Associated Class Type	GUI Component	Associated Class Type
Advertisement	`AdView`, `HtmlBannerWebView`, `AdContainer`	Layouts	`LinearLayout`, `AppBarLayout`, `FrameLayout`, `RelativeLayout`, `TableLayout`
Bottom Navigation	`BottomTabGroupView`, `BottomBar`	Button Bar	`ButtonBar`
Card	`CardView`	CheckBox	`CheckBox`, `CheckedTextView`
Drawer (Parent)	`DrawyerLayout`	Date Picker	`DatePicker`
Image	`ImageView`	Image Button	`ImageButton`, `GlyphView`, `AppCompatButton`, `AppCompatImageButton`, `ActionMenuItemView`, `ActionMenuItemPresenter`
Input	`EditText`, `SearchBoxView`, `AppCompatAutoComplete TextView`, `TextView`[1]	List Item (Parent)	`ListView`, `RecyclerView`, `ListPopupWindow`, `TabItem`, `GridView`
Map View	`MapView`	Multi-Tab	`SlidingTab`
Number Stepper	`NumberPicker`	On/Off Switch	`Switch`
Pager Indicator	`ViewPagerIndicatorDots`, `PageIndicator`, `CircileIndicator`, `PagerIndicator`	RadioButton	`RadioButton`, `CheckedTextView`
Slider	`SeekBar`	TextButton	`Button`[2], `TextView`[3]
Tool Bar	`ToolBar`, `TitleBar`, `ActionBar`	Video	`VideoView`
Web View	`WebView`	Drawer Item	`Others category and ancestor is Drawer(Parent)`
List Item	`Others category and ancestor is List(Parent)`	Others	`...`

[1] The property `editable` needs to be `TRUE`.
[2] The GUI component needs to have a non-empty `text` property.
[3] The property `clickable` needs to be `TRUE`.

Table 9.1: The 26 categories (including the "Others" category) of GUI class types we used in `Screen2Vec` and their associated base class names. Some categories have additional heuristics, as shown in the notes. This categorization is adapted from [183].

the hierarchical GUI view tree. For example, a GUI component has a distance of 1 to its parent and children and a distance of 2 to its direct siblings.

Linear Layers

At the end of each of the two levels in the pipeline, a linear layer is used to combine multiple vectors and shrink the combined vector into a lower-dimension vector that contains the relevant semantic content of each input. For example, in the GUI component embedding process, the model first concatenates the 768-dimensional text embedding with the 6-dimensional class embedding. The linear layer then shrinks the GUI component embedding back down to 768 dimensions. The linear layer works by creating 774×768 weights: one per pair of input dimension and output dimension. These weights are optimized along with other parameters during the training process, so as to minimize the overall total loss (loss function detail in Section 9.2.3).

In the screen embedding process, a linear layer is similarly used for combining the 768-dimensional layout embedding vector with the 64-dimensional GUI content embedding vector to produce a new 768-dimensional embedding vector that encodes both the screen content and the screen layout.

Text Embeddings

We use a pre-trained state-of-art Sentence-BERT model [238] to encode the text labels on each GUI component and the Google Play store description for each app into 768-dimensional embedding vectors. This Sentence-BERT model, which is a modified BERT network [69], was pre-trained on the SNLI [37] dataset and the Multi-Genre NLI [286] dataset with a mean-pooling strategy, as described in [238]. This pre-trained model has been shown to perform well in deriving semantically meaningful sentence and phrase embeddings where semantically similar sentences and phrases are close to each other in the vector space [238].

Layout Embeddings

Another important step in the pipeline is to encode the visual layout pattern of each screen. We use the layout embedding method from [66], where we first extract the layout of a screen from its screenshot using the bounding boxes of all the leaf GUI components in the hierarchical GUI tree, differentiating between text and non-text GUI components using a different color (Figure 9.2). This layout image represents the layout of the GUI screen while abstracting away its content and visual specifics. We then use an image *autoencoder* to encode each image into a 64-dimensional

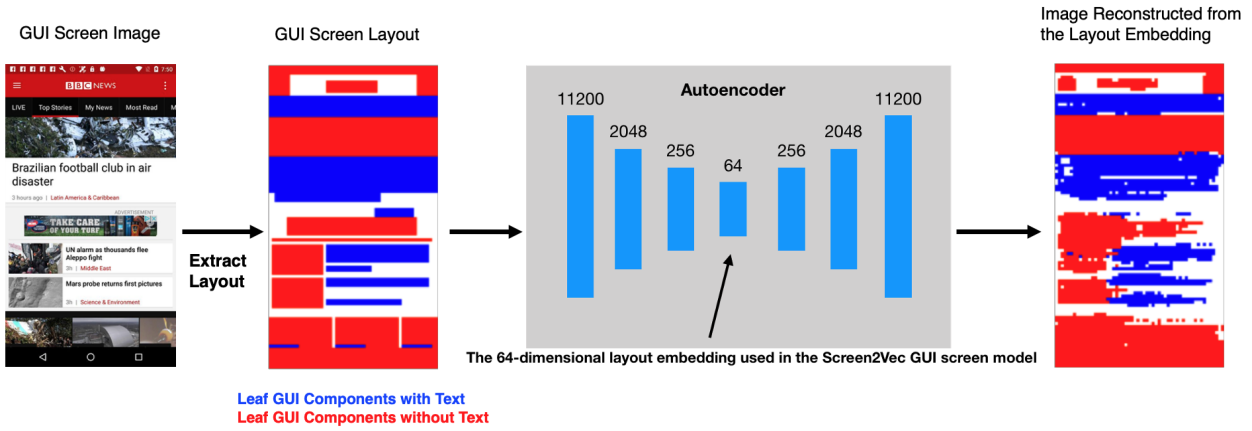


Figure 9.2: `Screen2Vec` extracts the layout of a GUI screen as a bitmap, and encodes this bitmap into a 64-dimensional vector using a standard autoencoder architecture where the autoencoder is trained on the loss of the output of the decoder [66].

embedding vector. The autoencoder is trained using a typical encoder-decoder architecture, that is, the weights of the network are optimized to produce the 64-dimensional vector from the original input image that can produce the best reconstructed image when decoded.

The encoder has input dimension of 11,200, and then two hidden layers of size 2,048 and 256, with output dimension of size 64; this means three linear layers of sizes $11,200 \rightarrow 2,048, 2,048 \rightarrow 256$, and $256 \rightarrow 64$. These layers have the Rectified Linear Unit (ReLU) [215] applied, so the output of each linear layer is put through an activation function which transforms any negative input to 0. The decoder has the reverse architecture (three linear layers with ReLU $64 \rightarrow 256, 256 \rightarrow 2,048$, and $2,048 \rightarrow 11,200$). The layout autoencoder is trained on the process of reconstructing the input image when it is run through the encoder and the decoder; the loss is determined by the mean squared error (MSE) between the input of the encoder and the output of the decoder.

GUI Embedding Combining Layer

To combine the embedding vectors of multiple GUI components on one screen into a single fixed-length embedding vector, we use a Recurrent Neural Network (RNN): The RNN operates similarly to the linear layer mentioned earlier, except it deals with sequential data (thus the "recurrent" in the name). The RNN we used was a sequence of the same linear layer, except with the additional input of a *hidden state*. The GUI component embeddings are fed into the RNN in the pre-order traversal order of the GUI hierarchy tree. For the first input of GUI component embedding, the hidden state was all zeros, but for the second input, the output from the first serves as the hidden state, and so on, so that the n^{th} input is fed into a linear layer along with $(n-1)^{th}$ output. The overall output is the output for the final GUI component in the sequence, which encodes parts of

all of the GUI components, since the hidden states could pass on that information. This allows screens with different numbers of GUI components to have vector representations that both take all GUI components into account *and* are of the same size. This RNN combiner is trained along with all other parameters in the screen embedding model, optimizing for the loss function (detail in Section 9.2.3) in the GUI screen prediction task.

9.2.3 Training Configurations

In the training process, we use 90% of the data for training and save the other 10% for validation. The models are trained on a cross entropy loss function with an Adam optimizer [135], which is an adaptive learning gradient-based optimization algorithm of stochastic objective functions. For both stages, we use an initial learning rate of 0.001 and a batch size of 256.

The GUI component embedding model takes about 120 epochs to train, while the GUI screen embedding model takes 80–120 epochs depending on which version is being trained[5]. A virtual machine with 2 NVIDIA Tesla K80 GPUs can train the GUI component embedding model in about 72 hours, and train the GUI screen embedding model in about 6-8 hours.

We used PyTorch's implementation of the `CrossEntropyLoss` function[6] to calculate the prediction loss. The `CrossEntropyLoss` function combines negative log likelihood loss (NLL Loss) with the log softmax function:

$$\begin{aligned} CrossEntropyLoss(x, class) &= NLL\_Loss(logSoftmax(x), class)) \\ &= -log(\frac{exp(x[class])}{\sum_c exp(x[c])}) \\ &= -x[class] + log\sum_c exp(x[c]) \end{aligned}$$

In the case of the GUI component embedding model, the total loss is the sum of the cross entropy loss for the text prediction and the cross entropy loss for the class type prediction. In calculating the cross entropy loss, each text prediction was compared to every possible text embedding in the vocabulary, and each class prediction was compared to all possible class embeddings.

In the case of the GUI screen embedding model, the loss is exclusively for screen predictions. However, the vector x does not contain the similarity between the correct prediction and *every* screen in the dataset; instead, we use *negative sampling* [201, 202] so that we do not have to recal-

[5]The version without spatial information takes 80 epochs; and the one with spatial information takes 120.

[6]`https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`

culate 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 having the model learn to differentiate the different screens within the same app.

9.2.4 Baselines

We compared `Screen2Vec` to the following three baseline models:

Text Embedding Only

The `TextOnly` model replicates the screen embedding method used in SOVITE 7. It only looks at the textual content on the screen: the screen embedding vector is computed by averaging the text embedding vectors for all the text found on the screen. The pre-trained Sentence-BERT model [238] calculates the text embedding vector for each text. With the `TextOnly` model, screens with semantically similar textual contexts will have similar embedding vectors.

Layout Embedding Only

The `LayoutOnly` model replicates the screen embedding method used in the original RICO paper [66]. It only looks at the visual layout of the screen: it uses the layout embedding vector computed by the layout autoencoder to represent the screen, as discussed in Section 9.2.2. With the `LayoutOnly` model, screens with similar layouts will have similar embedding vectors.

Visual Embedding Only

The `VisualOnly` model encodes the visual look of a screen by applying an autoencoder (described in Section 9.2.2) directly on its screenshot image bitmap instead of the layout bitmap. This baseline is inspired by the visual-based approach used in GUI task automation systems such as VASTA [247], Sikuli [299], and HILC [120]. With the `VisualOnly` model, screens that are visually similar will have similar embedding vectors.

[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.

9.2.5 Prediction Task Results

We report the performance on the GUI component task and the GUI screen prediction task of the `Screen2Vec` model, as well as the GUI screen prediction performance for the baseline models described above.

Table 9.2 shows the top-1 accuracy (i.e., the top predicted GUI component matches the correct one), the top-0.01% accuracy (i.e., the correct GUI component is among the top 0.01% in the prediction result), the top-0.1% accuracy, and the top-1% accuracy of the two variations of the `Screen2Vec` model on the GUI component prediction task, where the model tries to predict the text content for each GUI component in all the GUI screens in the RICO dataset using its context (the other GUI components around it) among the collection of *all* the GUI components in the RICO dataset.

Model	Top-1 Accuracy	Top 0.01% Accuracy	Top 0.1% Accuracy	Top 1% Accuracy	Top 5% Accuracy	Top 10% Accuracy
`S2V-EUCLIDEAN-text`	0.443	0.619	0.783	0.856	0.885	0.901
`S2V-HIERARCHICAL-text`	0.588	0.687	0.798	0.849	0.878	0.894

Table 9.2: The GUI component prediction performance of the `Screen2Vec(S2V)` model

Similarly, Table 9.3 reports the accuracy of the `Screen2Vec` model and the three baseline models (`TextOnly`, `LayoutOnly`, and `VisualOnly`) on the task of predicting GUI screens, where each model tries to predict each GUI screen in all the GUI interaction traces in the RICO dataset using its context (the other GUI screens around it in the trace) among the collection of *all* the GUI screens in the RICO dataset. For the `Screen2Vec model`, we compare two versions: one that encodes the locations of GUI components and the screen layouts and one that does not consider such spatial information. A higher accuracy indicates that the model is better at predicting the correct screen.

We also report the *normalized* rooted mean square error (RMSE) of the predicted screen embedding vector for each model, normalized by the mean length of the actual screen embedding vectors. A smaller RMSE indicates that the top prediction screen generated by the model is, on average, more similar to the correct screen.

From the results in Table 9.3, we can see that the `Screen2Vec` models perform better than the baseline models in top-1 and top-k prediction accuracy. Among the different versions of `Screen2Vec`, the versions that encode locations of GUI components and the screen layouts per-

form better than the one without spatial information, suggesting that such spatial information is useful. The model that uses the `HIERARCHICAL` distance performs similarly to the one that uses the `EUCLIDEAN` distance in GUI component prediction, but performs worse in screen prediction. In the Sample Downstream Tasks section below, we will use the `Screen2Vec-EUCLIDEAN-spatial info` version of the `Screen2Vec` model.

As we can see, adding spatial information dramatically improves the Top-1 accuracy (by 10x) and Top-0.01% accuracy (by 2x). But the improvements in Top 0.1% accuracy, Top 1% accuracy, and normalized RMSE are smaller. We think the main reason is that aggregating the textual information, GUI class types, and app descriptions is good for representing the high-level "topic" of a screen (e.g., a screen is about hotel booking because its text and app descriptions talk about hotels, cities, dates, rooms etc.), hence the good top 0.1% and 1% accuracy and normalized RMSE for the"no spatial info" model. But these types of information are not sufficient for reliably differentiating the different types of screens needed (e.g., search, room details, order confirmation) in the hotel booking process because all these screens in the same app and task domain would contain "semantically similar" text. This is why the spatial information is helpful in identifying the top-1 and top-0.01% results.

Interestingly, the baseline models beat the "no spatial info" version of `Screen2Vec` in normalized RMSE: i.e., although the baseline models are less likely to predict *the correct* screen, their predicted screens are, on average, more similar to *the correct* screen. A likely explanation to this phenomenon is that both baseline models use, by nature, similarity-based measures, while the `Screen2Vec` model is trained on a prediction-focused loss function. Therefore `Screen2Vec` does not emphasize making *more similar* predictions when then the prediction is incorrect. However, we can see that the `Screen2Vec-layout+locations` version of `Screen2Vec` performs better than the baseline models on both the prediction accuracy and the similarity measure.

Note that while the accuracy measures are indicative of how much the model has learned about GUI screens and components, the main purpose of the `Screen2Vec` model is *not* to predict GUI components or screens, but to produce distributed vector representations for them that encode useful semantic, layout, and design properties. Therefore we further evaluate `Screen2Vec` with several downstream tasks in the following Section 9.3.

9.3 Sample Downstream Tasks

This section presents several sample downstream tasks to illustrate important properties of the `Screen2Vec` representations and the usefulness of our approach.

Model	Top-1 Accuracy	Top 0.01% Accuracy	Top 0.1% Accuracy	Top 1% Accuracy	Top 5% Accuracy	RMSE
`S2V-EUCLIDEAN-spatial info`	0.061	0.258	0.969	0.998	1.00	0.853
`S2V-HIERARCHICAL-spatial info`	0.052	0.178	0.646	0.924	0.990	0.997
`S2V-EUCLIDEAN-no spatial info`	0.0065	0.116	0.896	0.986	0.999	1.723
`TextOnly`	0.012	0.055	0.196	0.439	0.643	1.241
`LayoutOnly`	0.0041	0.024	0.091	0.222	0.395	1.135
`VisualOnly`	0.0060	0.026	0.121	0.252	0.603	1.543

Table 9.3: The GUI screen prediction performance of the three variations of the `Screen2Vec` model and the baseline models (`TextOnly`, `LayoutOnly`, and `VisualOnly`).

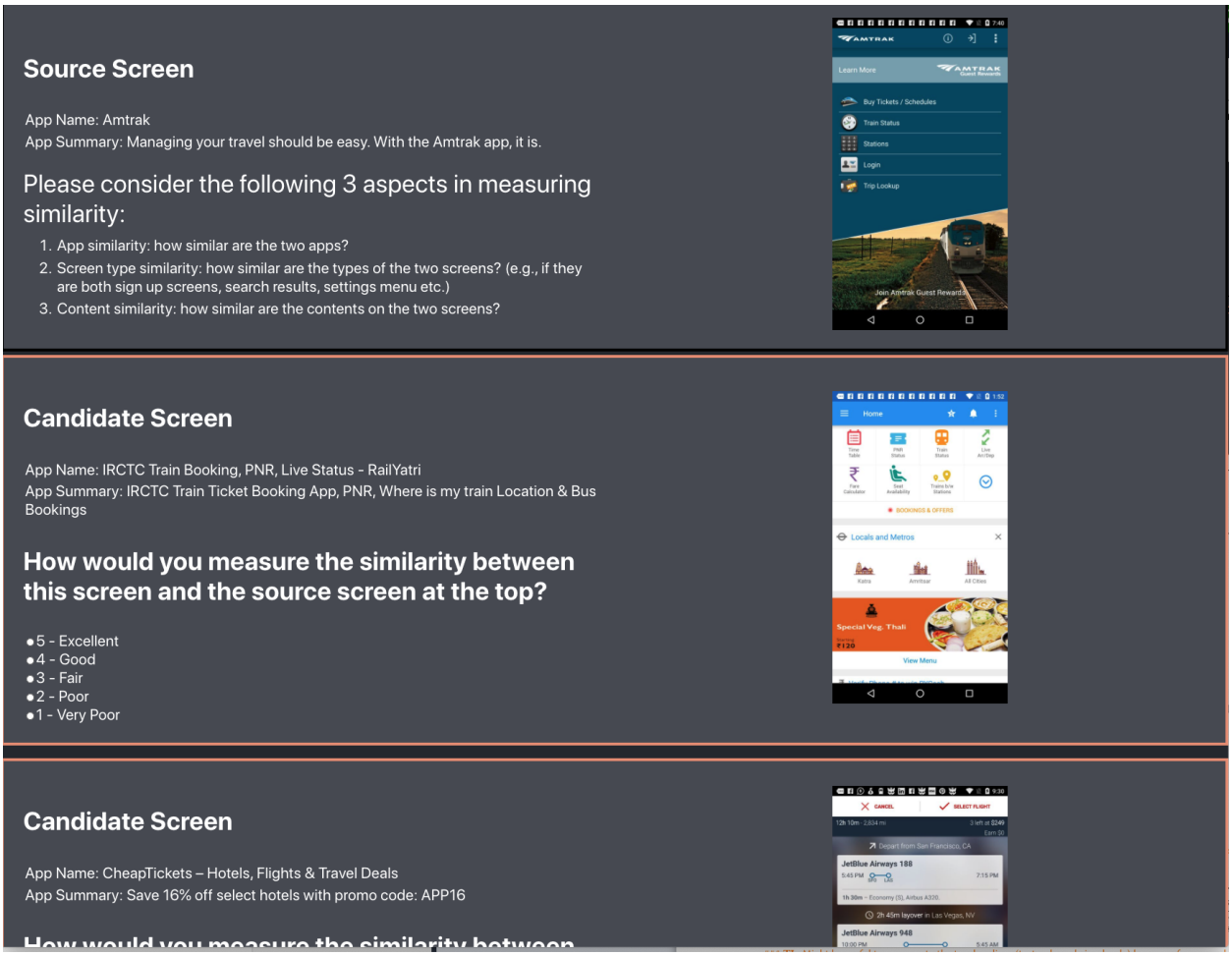


Figure 9.3: The interface shown to the Mechanical Turk workers for rating the similarities for the nearest neighbor results generated by different models.

9.3.1 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 [66]. 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? A 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 Mechanical Turk study to compare the similarity between the nearest neighbor results generated by the different models. We selected 50 screens from apps and app domains that most users are familiar with. We did *not* select random apps from the RICO dataset, as many apps in the dataset can be obscure to Mechanical Turk workers so they cannot understand them and therefore might not be able to judge the similarity of the results. For each screen, we retrieved the top-5 most similar screens using each of the 3 models. Therefore, each of the 50 screens had up to 3 (models) $\times$ 5 (screen each) = 15 similar screens, but many had fewer since different models may select the same screens.

79 Mechanical Turk workers participated in this study[8]. They labeled the similarity between 5,608 pairs of screens. Each worker was paid $2 for each batch of 5 sets of source screens they labeled. A batch on average takes around 10 minutes. In each batch, a worker went through a sample of 5 source screens from the 50 source screens in random order, where for each source screen, the worker saw the union of the top-5 most similar screens to the source screen generated by the 3 models in random order. For each screen, we also showed the worker the app it came from and a short description of the app from the Google Play Store, but we did not show which model produced the screen. The worker was asked to rate the similarity of each screen to the original source screen on a scale of 1 to 5 (Figure 9.3). We asked the workers to consider 3 aspects in measuring similarity: (1) app similarity (how similar are the two apps); (2) screen type similarity (how similar are the types of the two screens e.g., if they are both sign-up screens, search results, settings menu, etc.); and (3) content similarity (how similar are the content on the two screens).

Table 9.4 shows the mean screen similarity rated by the Mechanical Turk workers for the top-5 nearest neighbor results of the sample source screens generated by the 3 models. 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 `TextOnly` and `LayoutOnly` models. Tested with a non-parametric Mann-Whitney U test (because the ratings are not normally distributed), the differences between

[8]The protocol was approved by the IRB at our institution.

the mean ratings of the Screen2Vec model and both the TextOnly and the LayoutOnly models are significant ($p < 0.0001$).

Screen2Vec		TextOnly		LayoutOnly	
Mean Rating	Std. Dev.	Mean Rating	Std. Dev.	Mean Rating	Std. Dev.
3.295*	1.238	3.014*	1.321	2.410*	1.360

*$p < 0.0001$.

Table 9.4: The mean screen similarity rated by the Mechanical Turk workers for the top-5 nearest neighbor results of the sample source screens generated by the 3 models: Screen2Vec, TextOnly, and LayoutOnly.

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, Figure 9.4 shows the example nearest neighbor results for the "request ride" screen in the Lyft app. Screen2Vec model retrieves 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. With respect to the content and app domain, all of 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 a payment type, wallet, pay, add money), and textual strings that are semantically similar to "destination" and "trips" (e.g., drop off location, trips, bus, flights). But the model did not consider 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. As shown in Figure 9.4, 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 menu/information card at the bottom of the screen and the screen-wide button at the bottom of the menu.

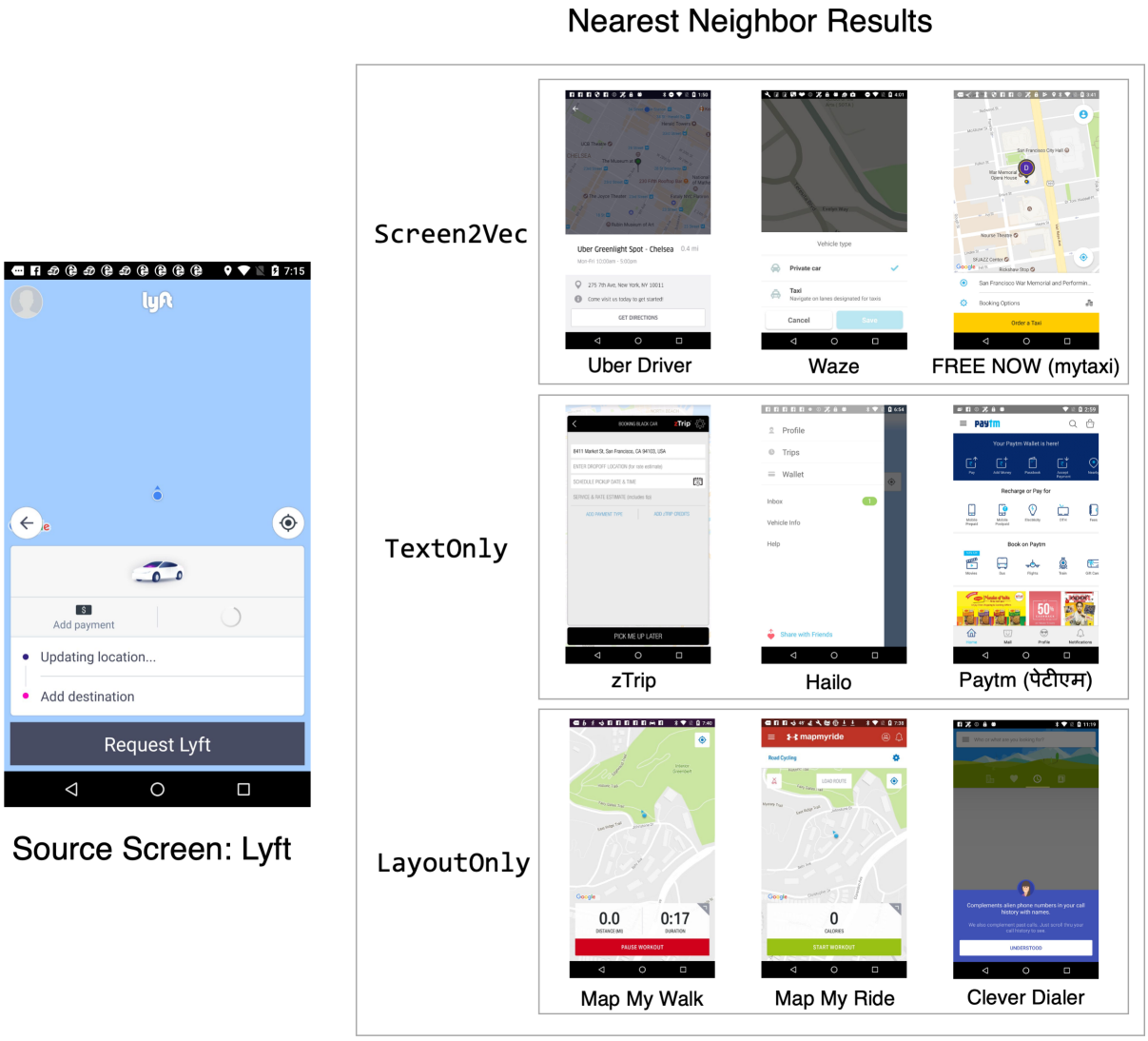


Figure 9.4: The example nearest neighbor results for the Lyft "request ride" screen generated by the `Screen2Vec`, `TextOnly`, and `LayoutOnly` models.

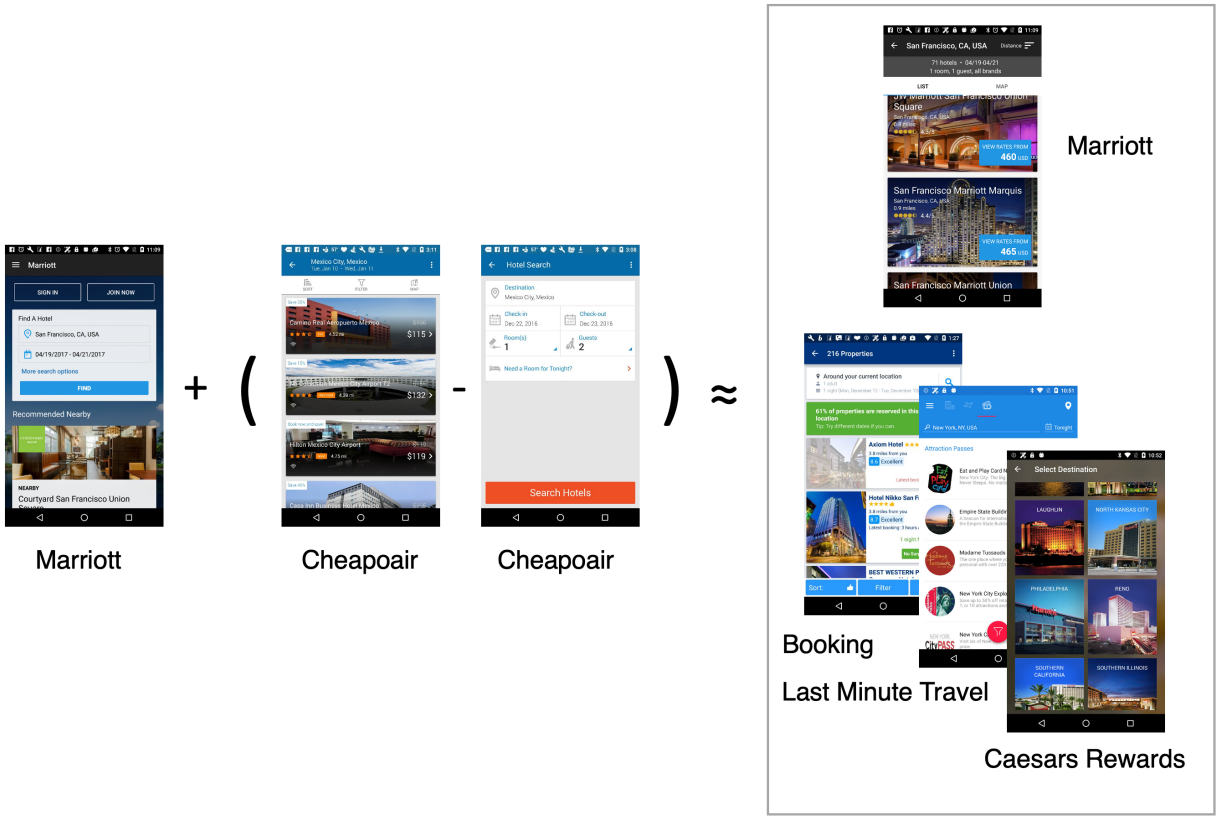


Figure 9.5: An example showing the composability of `Screen2Vec` embeddings: running the nearest neighbor query on the composite embedding of (Marriott app's hotel booking page + Cheapoair app's hotel booking page − Cheapoair app's search result page) can match the Marriott app's search result page, and the similar pages of a few other travel apps.

9.3.2 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. 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. We will discuss more its potential applications in Section 9.4.

9.3.3 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. In this section, we conducted a preliminary evaluation on the effectiveness of embedding mobile tasks as sequences of `Screen2Vec` screen embedding vectors.

Similar to GUI screens and components, the goal of embedding mobile tasks is to represent them in a vector space where more similar tasks are closer to each other. To test this, we recorded the scripts of completing 10 common smartphone tasks, each with two variations that use different apps, using the SUGILITE system on a Pixel 2 XL phone running Android 8.0. Each script consists of a sequence of "perform action X (e.g., click, long click) on the GUI component Y in the GUI screen Z". In this preliminary evaluation, we only used the screen context: we represented each task as the average of the `Screen2Vec` screen embedding vectors for all the screens in the task sequence.

Table 9.5 shows the 10 tasks we tested on, the two apps used for each task, and the number of unique GUI screens in each trace used for task embedding. We queried for the nearest neighbor within the 20 task variations for each task variation, and checked if the model could correctly identify the similar task that used a different app. The `Screen2Vec` model achieved a 18/20 (90%) accuracy in this test. In comparison, when we used the `TextOnly` model for task embedding, the accuracy was 14/20 (70%).

Task Description	App 1	Screen Count	App 2	Screen Count
Request a cab	Lyft	3	Uber	2
Book a flight	Fly Delta	4	United Airlines	4
Make a hotel reservation	Booking.com	7	Expedia	7
Buy a movie ticket	AMC Theaters	3	Cinemark	4
Check the account balance	Chase	4	American Express	3
Check sports scores	ESPN	4	Yahoo! Sports	4
Look up the hourly weather	AccuWeather	3	Yahoo! Weather	3
Find a restaurant	Yelp	3	Zagat	4
Order an iced coffee	Starbucks	7	Dunkin' Donuts	8
Order takeout food	GrubHub	4	Uber Eats	3

Table 9.5: A list of 10 tasks we used for the preliminary evaluation of using `Screen2Vec` for task embedding, along with the apps used and the count of screens used in the task embedding for each variation.

While the task embedding method we explored in this section 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, it would be useful 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.

9.4 Potential Applications

This section describes several potential applications where the new `Screen2Vec` technique can be useful based on the downstream tasks described in Section 9.3.

Screen2Vec can enable new GUI design aids that take advantage of the nearest neighbor similarity and composability of Screen2Vec embeddings. Prior work such as [66, 112, 140] has shown that data-driven tools that enable designers to curate design examples are quite useful for interface designers. Unlike [66], which uses a content-agnostic approach that focuses on the visual and layout similarities, Screen2Vec considers the textual content and app meta-data in addition to the visual and layout patterns, often leading to different nearest neighbor results as discussed in Section 9.3.1. 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) [150] and LayoutGAN [158] 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 Section 9.3.3, 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 Section 9.3.3 also indicated that Screen2Vec can embed mo-

bile 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' Donuts app).

9.5 Limitations and Future Work

There are several limitations of our work in `Screen2Vec`. First, `Screen2Vec` has only been trained and tested on Android app GUIs. However, the approach used in `Screen2Vec` should apply to any GUI-based apps with hierarchical-based structures (e.g., view hierarchies in iOS apps and hierarchical DOM structures in web apps). We expect embedding desktop GUIs to be more difficult than mobile ones, because individual screens in desktop GUIs are usually more complex with more heterogeneous design and layout patterns.

Second, the RICO dataset we use only contains interaction traces within single apps. The approach used in `Screen2Vec` should generalize to interaction traces across multiple apps. It would be good to evaluate its prediction performance on cross-app traces in the future with an expanded dataset of GUI interaction traces. The RICO dataset also does not contain screens from paid apps, screens that require special accounts/privileges to access to (screens that require free accounts to access are included when the account registration is readily available in the app), or screens that require special hardware (e.g., in the companion apps for smart home devices) or specific context (e.g., pages that are only shown during events) to access. This limitation of the RICO dataset might affect the performance of the pre-trained `Screen2Vec` model on these underrepresented types of screens.

A third limitation is that the current version of `Screen2Vec` does not encode the semantic information of graphic icons that have no textual information. Accessibility-compliant apps all have alternative texts for their graphic icons, which `Screen2Vec` already encodes in its GUI screen and component embeddings as a part of the text embedding. However, for non-accessible apps, computer vision-based (e.g., [53,183]) or crowd-based (e.g., [304]) techniques can be helpful for generating textual annotations for graphic icons so that their semantics can be represented in `Screen2Vec`. Another potentially useful kind of information is the rules and examples in GUI design systems (e.g., Android Material Design, iOS Design Patterns). While Screen2Vec can, in some ways, "learn" these patterns from the training data, it will be interesting to explore a hybrid approach that can leverage their explicit notions.

9.6 Related Work in Embedding Natural Language

The study of representing words, phrases, and documents as mathematical objects, often vectors, is central to natural language processing (NLP) research [202,272]. The conventional non-distributed word embedding method represents a word using a *one-hot* representation where the vector length equals to the size of the vocabulary, and only one dimension (that corresponds to the word) is on [272]. This representation does not encode the semantics of the words, as the vector for each word is perpendicular to the others. Documents represented using one-hot word representation also suffer from the curse of dimensionality [18] as a result of the extreme sparsity.

By contrast, a *distributed* representation of a word represents the word across multiple dimensions in a continuous-valued vector (word embedding) [20]. Such distributed representations can capture useful syntactic and semantic properties of the words, where syntactically and semantically related words are similar in this vector space [272]. Modern word embedding approaches are usually results from the language modeling task. For example, `Word2Vec` [202] learns the embedding of a word by predicting it based on its context (i.e., surrounding words), or predicting the context of a word given the word itself. GloVe [228] is similar to `Word2Vec` on a high level, but focuses on the likelihood that each word appears in the context of other words within the whole corpus of texts, as opposed to `Word2Vec` which uses local contexts. More recent work such as ELMo [229] and BERT [69] allowed contextualized embedding. That is, the representation of a phrase can vary depending on a word's context to handle polysemy (i.e., the capacity for a word or phrase to have multiple meanings). For example, the word "bank" can have different meanings in "he withdrew money from the bank" versus "the river bank"

While distributed representations are commonly used in natural language processing, to the best of our knowledge, the `Screen2Vec` approach presented in this chapter is the first to seek to encode the semantics, the contexts, and the design patterns of GUI screens and components using distributed representations. The `Screen2Vec` approach is conceptually similar to `Word2Vec` on a high level. Similar to `Word2Vec`, `Screen2Vec` is trained using a predictive modeling task where the context of a target entity (words in `Word2Vec`, GUI components and screens in `Screen2Vec`) is used to predict the entity (known as the continuous bag of words (CBOW) model in `Word2Vec`). There are also other relevant `Word2Vec`-like approaches for embedding APIs based on their usage in source code and software documentation (e.g., `API2Vec` [218]), and modeling the relationships between user tasks, system commands, and natural language descriptions in the same vector space (e.g., CommandSpace [1]).

Besides the domain difference between `Screen2Vec` and `Word2Vec` and its follow-up work, `Screen2Vec` uses both a (pre-trained) text embedding vector and a class type vector, and com-

bines them with a linear layer. It also incorporates external app-specific meta-data such as the app store description of the app. The hierarchical approach allows `Screen2Vec` to compute a screen embedding with the embeddings of the screen's GUI components, as described in Section 9.2. In comparison, `Word2Vec` only computes word embeddings using word contexts without using any other meta-data [202].

9.7 Chapter Conclusion

We have presented `Screen2Vec`, a new self-supervised technique for generating distributed semantic representations of GUI screens and components using their textual content, visual design and layout patterns, and app meta-data. This new technique has been shown to be effective in downstream tasks such as nearest neighbor retrieval, composability-based retrieval, and representing mobile tasks. `Screen2Vec` addresses an important gap in computational HCI research, and can be utilized for enabling and enhancing interactive systems in task learning (e.g., [167, 247]), intelligent suggestive interfaces (e.g., [52]), assistive tools (e.g., [27]), and GUI design aids (e.g., [149, 268]).

Chapter 10

Limitations, Future Work, and Conclusion

Each of the previous chapters in this dissertation has discussed some local limitations and future work individually. In this chapter, I summarize some of the most important ones in the entire SUGILITE project, plus more the more global ones for the scope of this dissertation.

10.1 System Limitations

10.1.1 Platform

SUGILITE and its follow up work have been developed and tested only on Android. SUGILITE retrieves the hierarchical 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 that to make it work with desktop apps would 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. In the current implementation of SUGILITE, there are also sometimes compatibility issues in non-native Android apps and apps that use special libraries or graphic engines to implement their GUIs (e.g., most games).

10.1.2 Runtime Efficiency

An important characteristic of SUGILITE is that it interacts with the underlying third-party apps in the same way that a human user does, 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 an excellent range 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 using SUGILITE is much slower than using 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. As discussed in Section 6.5, 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 [134]. Another relevant issue is that SUGILITE can only read and invoke items that are *logically* on the screen. Note that this is different from being *visibly* on the screen—for example, if a list contains 5 items but the screen can only display 3, the other 2 items may have been already rendered and "displayed" outside the visible area of the screen (depending on the implementation), which SUGILITE still have access to. However, if the app loads and renders additional menu items just-in-time when the user scrolls[1], *Sugilite* would not have access to the off-screen items. I will leave these issues for future work.

10.1.3 Expressiveness

My work in SUGILITE has made several contributions in improving the user expressiveness in programming by demonstration and interactive task learning systems. For example, the PUMICE interface described in Chapter 5 supports conditional (if-then) statements with user-taught recursively-defined concepts in the conditions. The EPIDOSITE mechanism described in Chapter 6 allows trigger-action structures with external triggers from app-displayed information, phone notifications, or events from external web services such as Gmail, Facebook, and Twitter. However, there are still several limitations in SUGILITE's expressiveness, which are left for future work.

[1]The default implementation of Android `ListView` does this with its view recycling mechanism.

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 complications 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. SUGILITE 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 ordering 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" screen (visually similar or having the same activity name), SUGILITE cannot know if the internal state of the underlying app has changed (e.g., adding the first item to the cart might affect 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.

10.1.4 Brittleness

While many measures (details described in Section 3.3.3) have been taken to help SUGILITE handle minor changes in app GUIs, SUGILITE scripts can still be brittle after a change in the underlying app GUI due to either an app update or an external event. As discussed in Section 4.3.1, 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 as some prior systems do, SUGILITE picks one or more features such as the text label, the ordinal position in a list (e.g., the 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.

However, if the GUI layout or structure has changed in any major way, the change will likely break the graph query, in which case the user can fix the script using the error handling mechanism

described in Section 3.3.3. The same applies to unexpected app states. If the app shows a GUI screen that has not been seen by SUGILITE during the demonstration (e.g., the surge price confirmation shows up in the Uber app when a price surge is currently in effect but the price was not in surge when the task was demonstrated by the user, or a new screen pops up for a new promotion), SUGILITE will enter the error-handling mode, asking the user to show how to handle the new situation (details in Section 3.3.3). In the future, it would be 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 (e.g., promos) without user intervention.

10.2 Future Work

10.2.1 Generalization in Programming by Demonstration

Generalization is a central challenge in programming by demonstration [65, 175]. 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 Chapter 9 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? Section 9.3.3 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, and Section 9.3.2 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. An interesting direction for future work is 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 of the user.

This approach is inspired by our observation about how people 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 instructions for 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 [36] and Atomic [242]. 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.

10.2.2 Human-Agent Co-Creation

The scope of SUGILITE falls into one type of human-AI collaboration where the user teaches the AI agent tasks that they have been already doing so that the agent can automate these tasks for them. In the future, I would love to explore the other type of human-AI collaboration, where the user "co-creates" new artifacts with the agent that do not yet exist and "co-performs" new tasks that the user is not performing yet on their own. The recent advances in the general-purpose language models (e.g., BERT [69], GPT-3 [41]) and reasoning models (e.g., COMET [36], Atomic [242]) are able to create "seemingly realistic" results from simple user inputs, showing great promise in this direction.

However, these models are still largely "black-boxes" to the user, leaving little agency to the user in controlling the output, limited transparency in helping the user understand the creation process, and lack of mechanisms in assisting the user to validate the model outputs. Many design implications from SUGILITE, such as its use of multi-modal interfaces for disambiguation, its mixed-initiative dialog approach for resolving vague and unknown concepts, its use of interactive visualization for communicating the system's state of understanding, and its application of familiar app GUIs for establishing common ground may also be useful in human-agent co-creation. The new generative models are great fits for few-shot learning and transfer learning, allowing effective incorporation of general knowledge with small amounts of data and instructions from end users. In SUGILITE, the strategy of using multi-modal interaction has been shown to be effective in helping users provide better-quality inputs to the system. Further research is needed on how users naturally select modalities in multi-modal environments, and on how interfaces can support more fluid transition between modalities. For the future, I plan to focus on designing the interaction

between the user and the AI system in these scenarios, bridging the transparency and explainability of AI systems with the user's natural cognitive models for the task, and enabling the user to provide useful inputs that boost the generalizability, expressiveness, robustness of the AI system through effective human-AI collaboration.

10.2.3 Field Study of SUGILITE

Another future direction is to study the user adoption of SUGILITE through a longitudinal field study. A field study was proposed as a part of this dissertation but had to be canceled due to the COVID-19 pandemic. 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 main goal of the deployment would be to study SUGILITE within its intended context of use [253]. Another goal is to study the use of SUGILITE in a more representative user populations. While our prior lab usability studies controlled for the programming expertise of the participants. Our participants are in general younger, with a higher education level, and come from a higher socioeconomic status than an average smartphone user. We seek to recruit a more diverse group of participants that can better resemble the wide range of smartphone users in the planned field study of SUGILITE.

10.3 Conclusion

In this dissertation, I have described the SUGILITE system and several of its extensions, new interfaces, and mechanisms I designed, developed, and studied to enable an intelligent task automation agent to learn new tasks and procedures interactively from the user's explicit instructions.

The GUI-mediated approach used in SUGILITE featured a new paradigm for interactive task learning. It has a much lower usability barrier for users than conventional programming languages or visual languages because users are already familiar with the GUIs of the existing third-party mobile apps. Compared with popular end-user development software such as IFTTT, SUGILITE's approach is not restricted to a small number of "supported task domains and devices" and supports a greater level of expressiveness for the users (as discussed in Chapters 3, 5, and 6). On the other end, compared with approaches that use *unconstrained* visual demonstrations or *unconstrained* natural language instructions, SUGILITE significantly lowers the technical difficulty for the computing system to understand the user's instructions by naturally constraining and ground-

ing the user's instructions to the existing app GUIs. This allows SUGILITE to avoid the grand AI challenge in recognizing, understanding, and reasoning with unconstrained natural language inputs and visual demonstrations in the physical world, which is extremely difficult for a computing system without extensive domain-specific training data.

SUGILITE's approach informs a new design strategy for human-AI interaction. To facilitate human-AI collaboration, a crucial step is to choose a *representation* in which both the human user and the AI system can effectively work. The human user should be able to easily and naturally express their intents in this representation, while the AI system should be able to understand user intents in this representation. Meanwhile, the AI system also needs to communicate its state of understanding in a representation which the human user can easily understand, process, and adjust their inputs or provide additional inputs accordingly (as discussed in Chapter 7).

My work in SUGILITE applied theoretical frameworks and design principles in multi-modal interfaces [222] and mixed-initiative interfaces [4, 111], adding to their implications for use in interactive task learning. Particularly, my work showed that the mutual disambiguation approach combining spoken natural language inputs with visual demonstrations is useful for generalization, disambiguation, error discovery, and error repair (as shown in Chapters 3, 4, and 7). The design of the APPINITE, PUMICE, and SOVITE interfaces demonstrated the application of several key design principles in mixed-initiative interaction such as each agent (human or computer) contributing what it is best suited at the most appropriate time to build mutual understanding, considering the uncertainty about the user's goals, and facilitating agent-user collaboration to refine results [111] in GUI-based interactive task learning.

On the AI aspect, my dissertation has made several contributions in new techniques for representing the semantics of app GUIs and grounding the natural language inputs to GUI demonstrations. Chapter 4 presented a novel method for representing a GUI screen as a graph and the technique for interactively parsing the user's intent for an action into its corresponding graph query through multi-turn conversations. Chapter 5 described a new lazy-evaluation dialog structure that allows the recursive clarification of ambiguous concepts and learning of unknown concepts. Chapter 7 presents a new multi-modal interface for error identification and recovery in human-AI interaction systems, which is a important but often overlooked topic in designing AI systems. The new privacy-preserving script sharing mechanism discussed in Chapter 8 not only facilitates script sharing between users but also enables us as researchers to collect GUI screen data and GUI interaction traces at a scale without accidentally collecting personal information from the users. Chapter 9 featured a new technique for generating semantic embeddings of GUI screens and com-

ponents that encodes their textual contents, visual design and layout patterns, and app meta-data without requiring manual data annotation.

As a whole, this dissertation represents an effort to empower non-technical users to teach new tasks, concepts, and their personal preferences and needs to computers through natural interactions. I hope work in this direction, including mine, can support a *bottom-up* approach in AI automation that helps individual users automate and augment their work. In my view, this is a key path towards alleviating the societal problem faced by the workforce in the era of "the fourth industrial revolution [245]". While the adoption of AI automation technologies has significantly increased the overall productivity of society, the gains have mostly gone to a small group at the top while many workers have been stagnated in wages or displaced by AI automation. I attribute much of this phenomenon to the *top-down* approach used in developing and deploying AI automation. While many workers, especially the rapidly expanding group of "gig workers", participate in the corporate-provided AI infrastructures, they often act as interchangeable and replaceable components (e.g., Uber drivers, Instacart shoppers, data annotators) in large algorithm-directed workflows with little agency. The AI systems are often developed from a corporate's perspective with its best interest in mind, with little support for individual worker's personal preferences, motivation, and creativity. Instead of replacing workers with AI, this end-user-programmable agent approach can emancipate workers from mundane repetitive tasks, enabling them to focus on the creative and social aspects of work which AI systems cannot feasibly replace in the near future. Bottom-up AI automation can complement the predominant top-down AI automation paradigm, empowering the users with a programmable AI agent with more agency, more control, more transparency, and more power.

Appendix A

A Need-finding Study for Understanding Text Entry in Smartphone App Usage

This appendix presents a need-finding study we conducted in 2015–2016 for understanding text entry in smartphone app usage. The results of this study informs and motivates the design of SUGILITE, especially on its support for the automation of cross-app repetitive mobile tasks.

Abstract

Text entry makes up about one-fourth of the smartphone interaction events, and is known to be challenging and difficult. However, there has been little study about the characteristics of text entry in the context of smartphone app usage. In this appendix, we present a mixed-method in-situ study of 17 active smartphone users to better understand text entry in smartphone app usage. Our results show 80% of text was entered into communication apps, with different apps exhibiting distinct usage patterns. We found that structured data such as URLs and email addresses are rarely typed but instead are auto-completed or replaced with search, copy-and-paste is rarely used, and sessions of smartphone usage with text entry involve more apps and last longer. We conclude with a discussion about the implications on the development of systems to better support mobile interaction.

A.1 Introduction

Smartphones are the primary devices on which users send and receive messages, take notes, and search for information on the go. Text entry is an important aspect of users' mobile interactions

and makes up about one-fourth of the total interaction events according to our data (with the other three-quarters being taps, swipes, focuses and other interactions). It is also long been known that text entry is a particularly challenging and difficult activity in using mobile phones [139, 187, 280].

Many solutions have been proposed to improve the performance and user experience of mobile text entry, including novel (virtual) keyboards (e.g., [302]), alternative non-keyboard text entry techniques (e.g., [289]), text prediction (e.g., [281]), and apps to support a specific domain of text entry (e.g., [47]). There also have been text corpora which have been collected from a single domain of smartphone usage, such as for email [279] or chat rooms [302], which have been used to help evaluate text entry methods. However, surprisingly little study has focused on understanding users' text entry in the context of smartphone app usage. This lack of knowledge makes it hard to properly evaluate if a proposed solution to text entry problems is relevant. It also hinders the development of new text entry mechanisms, as researchers often need to make guesses or assumptions about users' text entry behaviors.

To help fill in this gap, we conducted a mixed-method study including surveys, interviews, diaries and in-situ logging to characterize the users' mobile text entry behaviors. In our study, we recorded 1,607,199 total events, of which 401,875 were text-editing events, representing 476,322 characters entered into 155,300 app usages of 364 unique apps from 17 active smartphone users (mostly university students) over 2 weeks. We also gathered the participants' reflections on their usages through interviews and diaries.

Using the collected data, in this appendix, we mainly focus on answering the following research questions:

RQ1: To which applications are data entered into smartphones?

RQ2: What kinds of textual data are entered into smartphones?

RQ3: Through what means are data entered (typing vs. copy-and-paste vs. auto-completion)?

RQ4: How are text-entering sessions different from the non-text entering sessions?

RQ5: How is text transferred among apps within sessions?

In summary for the above questions, we found that for our users, over 80% of text was entered into communication apps like instant messaging (IM), SMS and email. Among those apps, the participants exhibit different text entry patterns: only a small portion of email app usages involve text entry compared to IM and SMS, but more email app usage involves heavy text entry (¿100 characters) than the other two communication apps.

Only a small portion (2.5%) of the strings entered into the smartphones by our participants includes structured data in the form of phone numbers, email addresses, URLs or street addresses, and most of them go to where one might expect (e.g., URLs are mostly entered in a browser). In addition, for about half of the times that the clipboard was used, the clipboard contents contained structured data, mostly URLs.

We found that in our data, 60% of the text was typed character by character into phones. Most of the rest were auto-completed, with less than 1% being entered through copy-and-paste. Looking at the various sessions, we found that sessions with text entry on average last longer and have more apps used than those with no text entry. Our interview results showed that most of our participants found copy-and-paste difficult to use and preferred alternative ways of sharing data, such as using the Android "share to" API, or taking a screenshot of an app and sharing it with other people.

While our study of smartphone usage behavior was performed on a sub-population of users for a limited period of time, and the results will not necessarily generalize across all different populations, our findings provide value, as discussed in [58], in uncovering interesting user practices and providing insights into how future smartphone interactions and apps can better support text entry and thus improve the overall user experience.

A.2 Related Work

There have been prior studies that tried to understand various aspects of smartphone usage. Böhmer et al. [42] ran a large-scale logging study of app launches and found that the use of an app on average lasts less than a minute. They also discovered that communication is the most often used app category, with some app categories being used more often at certain times. The LiveLab project [249] also collected a large dataset of smartphone app usage records. Many prior studies (e.g., [15, 40, 59, 128, 270]) investigated how smartphone usage is triggered by the context and how smartphone interactions can be integrated with the user's activity. Karlson et al. [129] looked at smartphone use in the context of task flow, suggesting that interruptions affect mobile productivity and found it common for users to move to a PC to complete a task started on the phone. Some studies also focused on characterizing users' information needs: Sohn et al. [254] conducted a diary study and discovered that people use diverse and sometimes ingenious ways to obtain information on smartphones.

To fully understand smartphone usage, it is not sufficient to only analyze the use of individual apps, as each smartphone app is designed to handle a limited number of domains (usually one), which is not sufficient to support many cross-domain tasks [264, 266]. Thus, many researchers

have looked at multi-app usage sessions. For instance, for mobile search tasks, Carrascal and Church [43] reported that users frequently switch among mobile search apps and other apps, and that sessions with mobile searches also tend to have more apps used, have a longer duration; and use some categories of apps more intensively. On the other hand, many smartphone usage sessions can be very short with studies reporting that 40% of sessions are shorter than 15 seconds [83] and 38% contain no app usage but only interactions with the lock screen or launcher [13] (for example, to look at notifications or the time). Jesdabodi and Maalej [123] proposed the concept of "usage states" to categorize usage sessions based on the topic of user's task. Many projects (e.g., [126, 174, 225, 250, 256, 294, 309]) investigated the "app chain" – the temporal patterns in app launches or the contextual information of the smartphone usage to predict the next app to be launched. Systems like HELPER [265] and SUGILITE [159] proposed using intelligent agents to automate tasks across multiple apps and domains.

However, none of the above studies looked at the text entry aspect of smartphone usage, nor is there any dataset for text entry behaviors in app usage available to the community. There were text corpora in some prior projects [259, 279, 302] collected from users' mobile text entry in a single domain (like emails or chat rooms) for the purpose of training and evaluating the performance of text entry methods. But they did not consider text entry behaviors in the context of multiple mobile app interactions, nor did they characterize the text entry behaviors so we can learn what data do users enter or what apps are these data entered into. There are also studies on the usage of a single domain of app with heavy text entry like instant messaging [57, 71] and SMS [17].

Various systems have been developed to better support mobile text entry. Some [47, 48, 217] support easier and more efficient entry and processing of structured data like dates, addresses and URLs. Fuccella et al.'s work [87] supports mobile text editing with gestures. Although there are many solutions proposed, there are few studies about how often those solutions will be relevant, or how prominent the reported problems are, as we do not yet know in which apps do users enter and process structured data, nor do we know how those data are entered and what types of structured data are common. We hope our results reported here can help the community to have a deeper understanding of users' text entry behaviors with respect to app usage and usage sessions, and provide resources for the development of future systems to better support smartphone text entry.

A.3 Pre-Survey

To help us better design our studies, we conducted an online survey using SurveyMonkey about the users' text entry and cross-app usage on their smartphones. By posting the link on Facebook

groups and to mailing lists at Carnegie Mellon University, we collected 65 responses between December 2015 and January 2016. 70% of the respondents were between 18–24 years old, 87% were students, 57% used iPhone while the rest (43%) used Android phones. In the survey, we asked the respondents to estimate their frequency of entering text for different purposes on their smartphones, and to describe the tasks, the goals, the procedures and the apps used for each of their most recent text entry situations. We also asked about their copy-and-paste usage, retyping, cross-application interactions and repetitive sequences of actions.

When asked about *which* applications they most enter data into on their smartphones, 76% of the respondents self-reported that they often entered data into map/navigation apps, 70% said they often entered data into instant messaging apps, and 66% said search engines. On the other end of the spectrum, most people reported that they never or seldom entered data into spreadsheets (58% of the respondents said never; 30% said seldom), text editing apps (30% never; 50% seldom) or note-taking apps (10% never; 53% seldom) on their smartphones.

For copy-and-paste, 3% respondents reported that they never used copy and paste, 12% seldom used it, 52% some-times used it and 33% often do so. Among the 53 responses who reported about their most recent copy-and-paste usage, the contents reported by 25 people (47%) consisted of "structured text," which we defined in the survey as URLs, phone numbers, email addresses and street addresses. The most popular type of structured text was URL, which was reported by 15 people (28%). Respondents also shared their most recent case of retyping contents instead of using copy-and-paste, mostly because they said that the text field did not allow copy-and-paste, or that copy-and-paste is too hard to use. For example, as a rationale for not using copy-and-paste, some of our respondents wrote:

> *"I saw it [the text] on a link and didn't want to accidentally click on the link."*
>
> *"... It was that either it was ineffective to try to copy paste due to resolution, clutter of other text in the browser or the address as typed didn't work on Google Maps."*
>
> *"The input box doesn't support paste".*

The respondents also shared their most recent experiences of using multiple apps for a single task. Sample usage scenarios they described included:

> *"Plan the event using email or messenger and check whether I'm free using Calendar."*
> *"[I used] Airbnb for the address, looked up the route through Google Maps, checked the bus schedule with [the] browser."*

> *"I opened Gmail to locate the receipt, I noted the total amount I had spent and divided it fairly between my friends (evenly, in this case). I then opened Venmo and charged each friend who owed me separately."*
>
> *"Find the name in Yelp, check rating and reviews, and then find the estimated time it takes from my place to the bar by using Google Maps."*

31 respondents were able to describe their most recent experiences of performing a repetitive task. A respondent talked about how she kept all her expenses in memos and she "saw the money I spent each day on notes and then added each entry on calculator." One other talked about how he needed to switch back-and-forth between the Cisco tool, system settings and the Browser to connect to the office network every morning.

The results of the pre-survey further suggest that users exhibit different text entry behaviors in different domains, and the amount of text, the frequency of text entry and the type of content entered vary for different categories of apps. It also suggests that in many situations, users need to use multiple apps, sometimes with data being transferred among the apps, to perform a single task. The outcome of the pre-survey shows the need for a more detailed empirical study to better understand users' text entry and cross-application behaviors, which motivates the main study we will talk about in the following sections.

A.4 Longitudinal Study

We conducted a two-week longitudinal study with 17 active smartphone users in February and March of 2016 to understand the users' data entry and cross-application usage in depth. The study consisted of two in-lab user interviews and a 2-week-long in-situ logging and diary study. We collected detailed logs of the users' smartphone interactions (clicking, tapping etc.), data entry behaviors (keystroke, copy-and paste), app usage and the corresponding usage context using a tracking tool that we created and installed on their phones. Through surveys and interviews, we also gathered the users' reflections on their smartphone interactions and descriptions of their specific usage sessions.

A.4.1 Participants

We recruited 17 participants (7 males and 10 females, ages 18–45, mean age 25.2, SD=5.48) from Carnegie Mellon University. All participants were proficient in English and used English as the main language in their interactions with smartphones. All participants use Android phones running

Android 4.4 or above as their main mobile phones, and they all self-identified as active smartphone users. During the course of study, the participants on average spent 4.2 hours on their smartphones per day, with the least active participant spending an average of 1.1 hours per day and the most active one spending on average 7.7 hours per day, so it seems reasonable to consider all of them to be active users. Each participant signed the informed consent form, and received a compensation of $50.

A.4.2 Tracking Tool

We implemented an Android app named CROSSAPPTRACKER to track the participants' smartphone interactions. CROSSAPPTRACKER runs as an Android accessibility service and records all of the participants' interactions with the phone UI across all smartphone apps. Each interaction (tapping, clicking, keystroke, focusing, clipboard event and screen state change) is recorded as a tracked event. Tracked events include when the user clicks or focuses on a UI element, enters or edits some text, when the window state changes, or when the contents of the clipboard changes. For each tracked event, the tracker also logs the name of the current app, the name of the current Android activity[1], the text or content description[2]of the UI element, a unique identifier of the UI element, a timestamp and the identifier (SSID) of the connected Wi-Fi network. In addition to these events, CROSSAPPTRACKER also records an event whenever the screen was turned on or off. CROSSAPPTRACKER automatically uploads the logs every 8 hours to our server, so the experimenter could look for interesting interactions. If the upload failed, CROSSAPPTRACKER would retry later to ensure the completeness of the log.

To protect privacy, CROSSAPPTRACKER allows the participants to specify a list of apps to exclude from keystroke logging, for example those that contain sensitive data, like banking apps and medical apps. It also avoids logging any data marked as encrypted or entered into the password fields. Still, the logs do contain sensitive data, so they are kept in a protected place and processed in accordance to our IRB approved study protocol.

A limitation of CROSSAPPTRACKER is that it can only listen to interaction events in native Android apps, but not when the user uses a web application. However, we do not expect this limitation to have a major impact on our results, as all but one of our 17 participants reported that they would almost always prefer using a native Android app to a web app when both are available.

[1]An activity is a single, focused thing that the user can do [95] within an Android application. Each activity often represents a single screen with a user interface.

[2]Content description is an accessible text label for a UI element that can be read out by a screen reader.

A study from Yahoo also reported that 90% of users' time on smartphones is spent in apps, while only 10% is spent in the browser [133].

A.4.3 Study Procedure

The study consisted of four main components:

Pre-interview

In 30-minute semi-structured interview sessions, the participants were asked to reflect on their smartphone usage in a one-on-one lab interview setting. They described the motivations, goals and procedures of concrete examples of their smartphone tasks that are cross-app, text entry heavy or repetitive. They also discussed the apps they used the most, how they split and coordinate tasks between their smartphone and other computing devices (computers, tablets, wearables), how they interact with the smartphone's virtual assistants (e.g., Google Now, S Voice), how they use copy-and-paste on their phones, and any smartphone tasks they find frustrating to perform, or would like to automate.

In-situ Logging

During the pre-interview, we installed CROSSAPPTRACKER onto the participants' smartphones. CROSSAPPTRACKER recorded all of the participants' smartphone interactions as discussed above, and uploaded the logs to our server, so the experimenter could look for interesting interactions and ask the participants to reflect on those in their daily diaries.

Each participant was asked to keep CROSSAPPTRACKER running in the background at all times for at least 14 days while using their smartphones as they normally would. Some participants recorded more than 14 days of data due to the scheduling of interview sessions, since the CROSSAPPTRACKER software was removed at the exit interview.

Diary Study

At the end of each day during the 2-week in-situ study, each participant filled out an online questionnaire about their smartphone usage on that day. Each questionnaire consisted of 4 questions. The first 3 questions were constant. The last one was a personalized question made by the experimenter about a specific usage session from that day. The 3 constant questions asked the respondents to describe the goals, procedures and contexts of their text entry heavy, cross-app or repetitive tasks on that day, if any. The personalized question was often in the form "we saw you used [list of apps]

at [time] today, can you report what was the task and how did you use the above apps to perform the task?"

Exit Interview

Each of the participants came into our lab for a 30-minute semi-structured interview session after completing the in-situ logging and diary study. During the session, the participants had their CROSSAPPTRACKER removed, and they were asked to clarify any confusing entries in their diaries and reflect on their smartphone usage of the past two weeks.

A.4.4 Dataset

The raw dataset we collected from the 17 participants through the in-situ logging contained 1,607,199 tracked events, representing 155,300 application usages in 37,496 sessions. An application usage contains all the tracked events starting from when an app became active in the foreground until either another app became active instead, or the phone was locked or turned off. Similarly, a session includes all the application usages starting from when the phone was unlocked or turned on until when the phone was locked or turned off. In the dataset, the 17 participants used 364 unique apps over the course of our study.

A.5 Results

A.5.1 Characterizing Text Entry Behaviors in App Usage

In this section, we mainly try to understand the characteristics of the users' text entry behaviors. More specifically, we attempt to answer three research questions:

RQ1: To which applications are data entered into smartphones?

RQ2: What kinds of data are entered into smartphones?

RQ3: Through what means are data entered (typing vs. copy-and-paste vs. auto-completion)?

Our dataset contained 401,875 text-editing events (addition, deletion or replacement of characters) made by the 17 participants in 348 different applications over roughly 2 weeks, representing a total of 476,322 characters entered, which works out to an average of 28,018 characters per person (SD=20802.5). Excluding symbols and spaces, 357,306 characters make up 71,974 words, with

an average word length 4.96. All the characters constitute 12,504 separate strings entered, or 52.5 strings per participant per day.

We group all the text-editing events into the corresponding app that they belong to, and use the app usage as our unit of analysis. Among all 155,300 app usages, 20,202 (13%) include at least one text entry event.

N	%Text-Session	Duration-Mean	#Char/Text-Session	%TextEntry	%Structured
20, 202	13.0%	27.9s (SD=80.6s)	23.6 (SD=51.0)	5.25%	2.50%

Keys:
%TextSession: Percentage of text entering app usages in all app usages
DurationMean: Mean duration for each text entering app usage
#Char/TextSession: Mean number of characters entered for each text entering app usage
%TextEntry: Percentage of *text heavy* app usages (with ¿100 characters entered) in all text entering app usages
%Structured: Percentage of strings containing structured data in all strings entered

Table A.1: Descriptive Statistics for App Usage

In Table A.1, we show the descriptive statistics for the 20,202 app usages with text-editing events (which we will call text entering app usages). For the analysis, we consider an app usage to be "text entry heavy" if more than 100 characters are entered into the app during this app usage. We also parsed the contents of the text entered and consider a piece of text to be "structured" if it contains an email address, a URL, a phone number or a US-formatted street address.

One thing we notice in our dataset is that there are distinct differences in the amounts of text entry among participants. The 5 participants with the smallest number of characters only entered on average 8,731 characters (SD=1519.7) during the study, while the 5 participants at the other end of the spectrum entered an average of 50,253 characters (SD=12789.8). With respect to app usages, the participants also exhibited diverse text entry behaviors: the bottom 5 participants (which are different than for characters entered) on average entered 12 characters (SD=3.5) per text entering app usage, while the top 5 on average entered 51 characters (SD=9.3). An interesting fact is that the 2 participants with the least characters entered per text entering app usage actually had the largest total number of text entering app usages. This suggests that users exhibit very different text entry patterns, with some entering a large number of shorter strings and some entering fewer, but longer strings.

We find no statistically significant difference between the two gender groups of our participants in the average amount of text entered per person (Male 28038.1, Female 28005.5), average number

of text entering sessions per person (Male 1280.9, Female 1123.6) or the mean session duration (Male 29.9s, Female 30.9s).

Category	Usage#	Example Apps
Communication-IM	26,178	WhatsApp, Facebook Messenger
Tools-Other	25,173	Google Search, Clock, Calculator, Dictionary, File Manager
Social	7,594	Facebook, Twitter, Pinterest
Communication-Browser	4,820	Chrome, Samsung Browser, Dolphin Browser
Communication-Email	3,633	Gmail, Inbox by Gmail, Yahoo! Mail
Communication-SMS	3,446	Messaging, Verizon Messages
Tools-Dialer	2,466	Phone, Dialer
Transportation	1,302	Uber, Lyft, RideTrack
News & Magazine	996	BaconReader for Reddit, NYTimes, Flipboard, Feedly
Media & Video	963	Youtube, Gallery, VLC
Music & Audio	852	Spotify, Google Music
Communication-Others	834	Contacts, Visual Voice Message, TrueCaller
Travel & Local-Maps	744	Google Maps, Waze, HERE Maps
Productivity-Others	700	Calendar, Adobe Reader, IFTTT, Dropbox
Photography	617	Camera, Google Photos, Pixlr
Entertainment	436	Netflix, 9GAG, IMDb
Productivity-Text Editor	422	Google Docs, POLARIS Office, Hamcom Office
Finance	403	Square, TurboTax, Venmo
Health & Fitness	366	Fitbit, S Health, Nike+
Lifestyle	362	Starbucks, Tinder, Zomato
Productivity-Memo	324	Evernote, Keep, Samsung Memo

Table A.2: App Category and Example Apps Sorted by Usage

RQ1: To which applications are data entered into smartphones? To answer RQ1 on understanding where the data are entered into smartphones, we logged the smartphone apps in which each text entry is entered, and group these apps using the categories from the Google Play store. We also further divide the text entry heaviest categories into sub-categories to better characterize the users' text entry behaviors in those apps. In Table A.2, we list the top 20 app categories with the most app usages and example apps within the category. Note that we have excluded system services and launchers from this table. Among all the text entered, 92.6% of the total 476,322 characters were entered into the top five most text-heavy categories: Communication-IM (64%), Communication-SMS (17%), Social (4.6%), Communication-Browser (4.0%) and Communication-Email (3.1%). All of the above five except for Email are also among the categories with the most number of text entering app usages. Among all the 20,202 text entering app usages, 12,477

(61.8%) are for IM, 1,588 (7.9%) are for SMS, 1,185 (5.9%) are for Tools-Other, 970 (4.8%) are for Browsers and 822 (4.1%) are for Social.

An interesting observation is that, within the Communication category, the subcategories exhibit very different profiles for the number of characters entered per text entering usage and the percentage of text entering usages in all usages. As shown in Figure A.3, for SMS and IM, 46.1% and 47.6% of usages respectively include text entry, but only 12.6% of email usages involve text entry. Among all the text entering usages, 14.9% SMS text entering usages are text-entry heavy (having 100+ characters entered), while the same statistic is 9.0% for email and 5.6% for IM.

This result implies that our participants are far less likely to send out or reply to emails on the smartphone compared to SMS and IM messages. This is consistent with what people said in our pre-interviews. When we asked our participants on how they would split tasks between smartphone and other computing devices, all of our participants said that they would not write long emails on their phones unless it was urgent. Instead, they would prefer to wait until they had access to a computer. About half of our participants reported that they seldom write any email on smartphones, mostly because "*it is hard and frustrating to type long emails on [a] phone.*" Others said they would reply to emails if it was quick, but would avoid writing formal or important emails because "*formatting emails on [a] smartphone is hard.*"We discover significant differences in the proportions of text heavy usage for SMS apps and IM apps (t=9,48, p¡.001). As shown in Figure 1, the percentage of text heavy usages is 2.7 times higher in SMS apps than in IM apps, which is interesting as we often consider these two apps to be used in similar scenarios. We asked some of our participants in the exit interview about how they would differentiate the use of SMS and that of IM, and got interesting insights. Three participants mentioned that the availability and easy accessibility of multi-media contents like pictures, video clips, animations (or "GIFYs"), emojis and audio messages in most IM apps reduce the need for typing long text messages. They said they would avoid typing long messages as much as possible and preferred to send the "alternative" form of messages because it was hard to type on phones. However, in situations like when the message recipient does not use IM apps, does not use the same IM app as the user does, or is not a friend on IM with the user, the user would have to use SMS to reach the recipient. This forces the user to type a longer message than they would when using IM apps due to the lack of easily accessible multimedia contents.

Other participants offered different perspectives on the rationale behind the text entry pattern difference between SMS and IM apps. One said that "*because [WhatsApp] is free, it incurs no extra cost for sending many shorter replying messages like 'Yeah', 'OK' or 'Sure'. But since I'm*

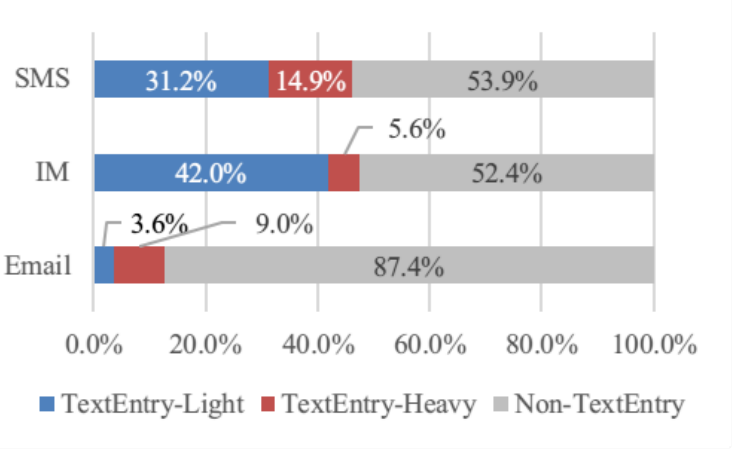


Figure A.1: Proportions of Light, Heavy and Non-Text Entry Usage for Three Types of Communication Apps

on a pay-by-use mobile plan, I need to pay extra to send SMS. That's why I send fewer SMS and try to cover more information in one message."

RQ2. What kinds of data are entered into smartphones? To investigate RQ2, we parsed and analyzed all 12,504 strings entered into the phones by our participants during the course of the study. We looked for structured data, including URLs, email addresses, phone numbers and street addresses. We were able to find 312 instances of strings with structured data, among which 72.8% include URLs, 27.6% include email addresses, 19.6% include phone numbers and 7.7% include street addresses. Note that the sum of counts for different types of structured data can exceed the total number of strings with structured data, as a string can contain more than one type of structured data, but we only count once if same type of structured data shows up more than once in a single string. We show the distribution of structured data entry usage among application categories in Figure 2 for categories with at least 5 strings with structured data entered.

This result confirms the hypothesis that the type of data entered is associated with the nature of the app [47]. Different types of structured text go to where you might think they should go—the participants entered URLs into browsers, email addresses into email clients, phone numbers into contacts and dialer apps, and street addresses into maps.

However, if we compare the number of structured data entered with the number of total text entering usages for each category of app, the portion of text entering usages with structured data entered is surprisingly small. As an example, for the browsers, only 12.2% of the text entering sessions had a URL entered. Similarly, only 11.6% text entering sessions for email apps had an

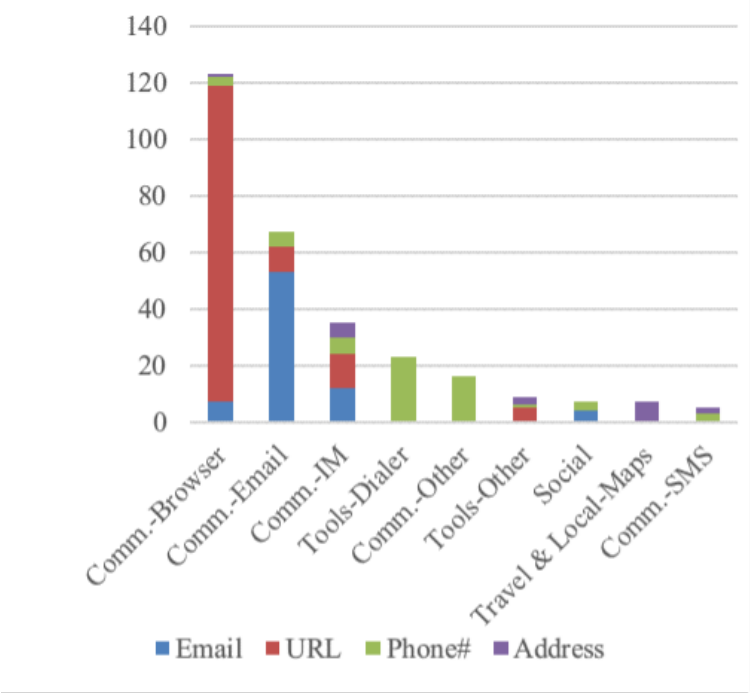


Figure A.2: Distribution of Structured Data Entry Usage across App Categories ("Comm" is the Communications category)

email address entered, 15.6% text entering sessions for dialer had a phone number entered and 5% text entering sessions for maps had a street address entered.

Talking with the participants in the exit interview, most of the participants reported that they seldom actually type structured data into any apps. The text fields in which one might expect structured data to be typed the most—the "To" fields in email clients, number fields in dialers, address bars in browsers or the address fields in map apps—are actually used as the de facto search bars in many modern app design, where the user can just type in the name (and in most cases, the partial name) of the contacts, web-sites or points of interest instead of the corresponding numbers, URLs or addresses. This technique reduced the user's typing workload and eliminated many needs for typing structured data.

Based on the results shown in Figure 2, for example, it is still safe to assume that users are more likely to type email addresses into email clients than into any other kinds of app, and to assume that users in email clients are more likely to type an email address than other types of structured data. But it would be wrong to assume that the user will type an email address most of the time when they send emails. The same goes for other combinations of structured data types and apps.

RQ3.Through what means are data entered? We then look at RQ3 on how the data are entered into the phones. Three major means of data entry are considered: typing, copy-and-paste

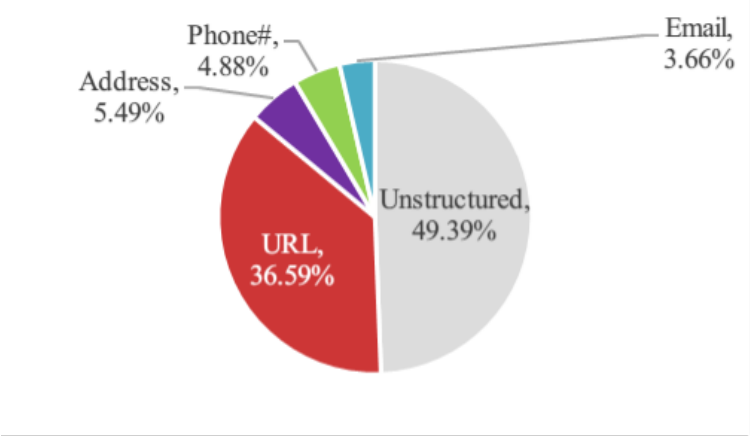


Figure A.3: Clipboard Contents by Structure Type

and auto-completion. A prior study found that not presenting the auto-completion suggestions enabled the users to type faster, but the suggestions were utilized to save taps and were subjectively preferred [236]. Copy-and-paste was also perceived to be difficult by users in a prior study [51].

In our 401,875 text-editing events collected, 51,534 (12.8%) were deletions, 311,369 (77.5%) were additions, in which 285,772 events were typing, 158 events were copy-and-pastes and 25,439 events were auto-completion. (For typing, each keystroke is a text-editing event. Each instance of paste or auto-completing is also an event.) The 12.8% of text entry events being deletions seems higher than other studies (e.g., 9.7% in [288] and 9.5% in [187]). Lengthwise, the 285,772 characters typed constituted 60.0% of the total 476,322 characters entered. 12,730 (2.7%) characters were pasted in, and the rest, 177,870 (37.3%), were entered using auto-completion or using a suggestion.

We investigated the characteristics of the content being copied into the clipboard. During the course of the study, 164 unique items were copied into the clipboard across all participants. That is only an average of 0.69 items per participant per day.

Among the texts that were copied, 50.6% contained structured data, among which the majority were URL links (72.3%). The distribution of content type for clipboard content is shown in Figure 3.

The low amount of clipboard usage confirms what our participants said in the pre-survey: that they do not use copy-and-paste often, as they find it frustrating. The participants also expressed the same frustration in the exit interview. When asked about the alternative means for transferring data between fields other than using copy-and-paste, we heard three major methods mentioned. For text that is short and easy to remember, many would simply retype. For sharing longer quotes, tweets or social network statuses with friends, a few of our participants mentioned that they often

just made a screenshot and sent the picture to their friends via IM apps. They considered using this technique to be quicker and easier, as taking a screenshot only requires pressing the power button and the volume-down button simultaneously—which is much easier to do than selecting the right block of text with one's fingers on the small screen. Text sent through screenshot can also be easier for the recipient to read as it preserves the format of the content.

In the exit interview, most users identified URLs as their most commonly copied structured data, which is consistent with our observations in the data. A participant reflected "*if I am to share a web content with my friends, I would normally just send a screenshot. But sometimes I'll copy-and-paste the link to the chat instead if the page is too long. Sometimes I'll also have to copy a link to my browser if it is not clickable.*"

N	#AppUsage-Mean	Duration-Mean	#UniqueApp-Mean	%Text-Entering	%MultiApp
17,684	4.76 (SD=8.64)	181.7s (SD=404.1s)	1.99 (SD=1.23)	39.9%	55.6%

Keys:
#AppUsageMean: Average number of non-system app usages in sessions
DurationMean: Mean duration for each session
#UniqueAppMean: Average number of unique non-system apps used in sessions
%TextEntering: Percentage of sessions with text entry
%MultiApp: Percentage of sessions containing more than one app usages

Table A.3: Descriptive Statistics for Sessions

A.5.2 Text Entry in Cross-application Interactions

In this section, instead of looking at the usage of individual apps, we focus on understanding text entry in the context of multi-app sessions. To characterize the user's text entry behavior in multi-app sessions, we investigated the following two research questions:

RQ4: How are text-entering sessions different from the non-text entering sessions?

RQ5: How is text transferred among apps within sessions?

As mentioned above, we define a *session* as containing all app usages from when the device is powered on or awakened until when the device is powered off or turned to standby mode. Based on this, we group 155,300 app usages in our dataset into 37,496 sessions. This definition of sessions is reasonable based on previous findings [277] that for the majority of instances where users return to their smartphone i.e., unlock their device, they in fact begin a new task as opposed to continuing

a previous one. Among our 37,496 sessions, 52.8% are turning on the phone display without launching any non-system apps, which are mostly to check the time or to see the notifications on the lock screen. We exclude those sessions in the following analyses. In Table A.3, we show the descriptive statistics for the 17,684 sessions with at least one app usage.

The statistics confirm phenomena reported by prior work. In Jesdabodi and Maalej's study [123], which used the LiveLab dataset [249], they reported a mean of 2.12 unique apps per session and overall mean duration of usage sessions of 172.8s, which are similar to our stats. We also find 58.9% of our sessions last less than 30s and 87.0% last less than 4 minutes, which supports Yan et al.'s finding [295] that 50% of phone engagements last less than 30 seconds, and 90% last less than 4 minutes. Our descriptive stats of sessions are also in line with Carrascal and Church's prior study [43] and confirms Ferreira et al.'s finding about mobile application micro usage [83].

	N	#AppUsage-Mean	Duration-Mean	#UniqueApp-Mean	%MultiApp
Non-Text	10,630	2.39 (SD=2.45)	129.7s (SD=339.9s)	1.66 (SD=0.93)	43.2%
Text	7,054	8.34 (SD=12.5)	260.0s (SD=474.4s)	2.49 (SD=1.44)	74.4%

Table A.4: Comparing Descriptive Statistics for Non-Text Entering Sessions and Text-Entering Sessions

RQ4: How are text-entering sessions different from the non-text entering sessions? Table A.4 shows a comparison of the descriptive statistics between the text entry sessions and the non-text entry sessions. We see that text entry sessions have higher average number of non-system app usages per session (t=39.5, p¡0.001), longer mean duration (t=19.9, p¡0.001), higher average number of unique apps in the sessions (t=42.8, p¡0.001) and higher percentage of multi-app sessions (t=44.1, p¡0.001).

RQ5: How is text transferred among apps within sessions? To answer RQ5, we consider two methods of data transfer between apps: copy-and-paste and the Android built-in "Share to" API.

One of the most obvious ways to transfer data between apps is through copy-and-paste. A prior study suggested that at least half of copy-paste operations were cross-app [51]. We discussed above the content of the clipboard and the low usage of copy-and-paste in RQ3 of the previous section. Here we focus on where are data copied from and to.

In Figure 4, we show the distribution of 164 copy events (including cuts) and 158 paste events across the app categories. 41% of the paste events are cross app, which is defined as that the

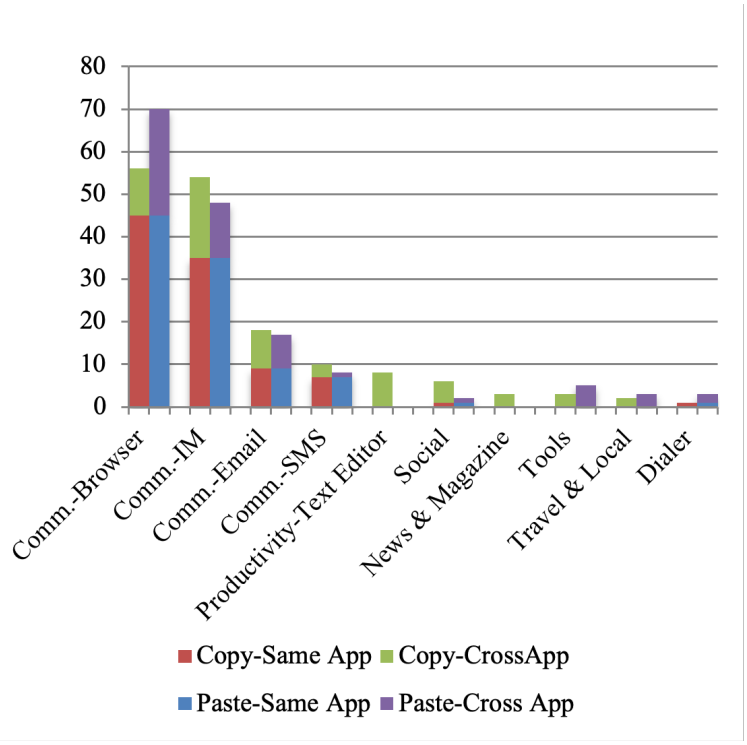


Figure A.4: Distribution of Copy-and-Paste Counts across App Catego-ries. "Copy-Same" means the copy or cut operation was in an app, and the corresponding paste was also in the same app. "Copy-CrossApp" means the copy/cut was in an app and the paste was in a different app. Similarly for "Paste-xx" where the paste was in this category and the copy/cut was either here or in a different app.

source of the clipboard content is different from the destination for pasting. The vast majority of copy-and-paste events (84.6%) are in the communication category, which is not surprising giving the dominance of communication app in the number of text entry sessions. The structure type of the pasted content is also associated with the app category. For instance, 67% of strings pasted into browsers include URL links, and 78% of strings with URLs in the clipboard are pasted into a browser.

Another method of data sharing across apps is through the Android "Share to" mechanism, which allows the user to share the current content to a set of apps that can "consume" this type of content. The developers for both the source app and the destination app have to explicitly specify the content of the data to share, the type of the data to share and the type of the data to accept. This mechanism is popular among our participants and covers many of the most frequent use cases like sharing a web page to WhatsApp chat, a piece of text to Gmail, etc.

We were not able to record the number of uses of the "share to" mechanism, since many mobile apps have overridden the default "share to" API provided by the Android operating system.

However, in the interviews, our participants mentioned they often use this mechanism since "*it better fits my way of thinking about the task. It's also much easier to use than copy-and-paste*".

A.6 Discussion

In the results, we see fewer instances of structured data entered than one might expect. From the interviews, we learn that this is mainly because in current app design, data entry fields can also be used as de facto search bars where the user can simply enter a partial entity name (e.g., the website name or recipient name) and the structured data (e.g., the URL or email address) will be retrieved. Such design is well used and liked by our participants.

Similarly, there are only a small number of copy-and-paste instances in our dataset. In interviews, the participants expressed their frustration about the current copy-and-paste mechanism, and talked about how they adopt alternative methods to transfer data between apps through the "screen-shot-and-send-picture" method, the Android "share to" mechanism, or app-specific ways of sharing data (e.g., through cloud service). This suggests that higher level support is needed to facilitate the data transfer between apps in cross-app tasks. For sharing web content with friends, the design for better supporting "sharing beyond the blue link", as framed in [23], is also worth exploring.

Our results support better integration and more streamlined data flow among apps. Users prefer to not type data into fields or copy-and-paste text between fields. Thus, they quickly adopt interactions techniques like integrated search and intent-based cross-app data sharing mechanisms. So, in the design of mobile interfaces, the designers should minimize the unnecessary or repetitive data entry, and support automatically sharing data from one app to another.

The results also show a correlation between the category of app in use and the type of structured data entered. This implies that predictive text entry methods should not only consider the language model and the semantic context, but also the context for the app usage.

A.7 Conclusion and Future Work

We conducted a mixed-method study on understanding user's text entry on smartphones in the context of app usage. Our results fill in gaps in prior work, identify characteristics of users' current text entry usage, and shed light on requirements for new systems. We characterize mobile text entry as to what, how, and to which apps the contents are entered, based on actual usage data

collected from our participants. We also consider text entry in the context of cross-app sessions and compare text-entering and non-text-entering sessions.

In this work, we studied the usage by a sub-population of smartphone users and provide insights on designing new novel interactions to better support their usage. As the next step, we plan to investigate text entry behaviors for populations with broader demographics, especially those from diverse socioeconomic backgrounds, people in developing countries [44], or field workers who have limited access to computers and use smartphone as their primary computing device [129]. Per our interviews, our participants seldom use voice input, but a future study should collect voice input data and compare how the usage, scenario and contexts of voice input differ from those of other text entry methods. We also hope to investigate a broader range of text entry, like text entry across multiple devices.

We also plan to perform an in-depth study about the relationships between data entry and the user's task flow, and investigate what we can infer about a user's other activities based on their text entry behaviors, and vice versa. Finally, we plan to use the resulting knowledge to guide the design of novel systems and interaction techniques to better support users' tasks on smartphones.

Bibliography

[1] Eytan Adar, Mira Dontcheva, and Gierad Laput. 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)*. ACM, New York, NY, USA, 167–176. DOI:http://dx.doi.org/10.1145/2642918.2647395

[2] Khalid Alharbi and Tom Yeh. 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)*. ACM, New York, NY, USA, 515–524. DOI:http://dx.doi.org/10.1145/2785830.2785892

[3] James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. PLOW: A Collaborative Task Learning Agent. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2 (AAAI'07)*. AAAI Press, Vancouver, British Columbia, Canada, 1514–1519.

[4] James F. Allen, Curry I Guinn, and Eric Horvtz. 1999. Mixed-initiative interaction. *IEEE Intelligent Systems and their Applications* 14, 5 (1999), 14–23.

[5] Amazon. 2020. Alexa Design Guide. (2020). https://developer.amazon.com/en-US/docs/alexa/alexa-design/get-started.html

[6] V. Antila, J. Polet, A. Lämsä, and J. Liikka. 2012. RoutineMaker: Towards end-user automation of daily routines using smartphones. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. 399–402. DOI:http://dx.doi.org/10.1109/PerComW.2012.6197519

[7] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A Survey of Robot Learning from Demonstration. *Robot. Auton. Syst.* 57, 5 (May 2009), 469–483. DOI:http://dx.doi.org/10.1016/j.robot.2008.10.024

[8] Zahra Ashktorab, Mohit Jain, Q Vera Liao, and Justin D Weisz. 2019. Resilient Chatbots: Repair Strategy Preferences for Conversational Breakdowns. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 254.

[9] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. *The semantic web* (2007), 722–735. http://www.springerlink.com/index/rm32474088w54378.pdf

[10] Amos Azaria, Jayant Krishnamurthy, and Tom M. Mitchell. 2016. Instructable Intelligent Personal Agent. In *Proc. The 30th AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 4.

[11] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. uLink: Enabling User-Defined Deep Linking to App Content. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 305–318. DOI:http://dx.doi.org/10.1145/2906388.2906416

[12] Bruce W. Ballard and Alan W. Biermann. 1979. Programming in Natural Language "NLC" As a Prototype. In *Proceedings of the 1979 Annual Conference (ACM '79)*. ACM, New York, NY, USA, 228–237. DOI:http://dx.doi.org/10.1145/800177.810072

[13] Nikola Banovic, Christina Brant, Jennifer Mankoff, and Anind Dey. 2014. ProactiveTasks: The Short of Mobile Device Use Sessions. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services (MobileHCI '14)*. ACM, New York, NY, USA, 243–252. DOI:http://dx.doi.org/10.1145/2628363.2628380

[14] Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 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)*. ACM, New York, NY, USA, 83–92. DOI:http://dx.doi.org/10.1145/2380116.2380129

[15] Louise Barkhuus and Valerie E. Polichar. 2011. Empowerment Through Seamfulness: Smart Phones in Everyday Life. *Personal Ubiquitous Comput.* 15, 6 (Aug. 2011), 629–639. DOI:http://dx.doi.org/10.1007/s00779-010-0342-4

[16] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 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)*. ACM, New York, NY, USA, 748–764. DOI:http://dx.doi.org/10.1145/2983990.2984020

[17] Agathe Battestini, Vidya Setlur, and Timothy Sohn. 2010. A Large Scale Study of Text-messaging Use. In *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '10)*. ACM, New York, NY, USA, 229–238. DOI:http://dx.doi.org/10.1145/1851600.1851638

[18] Richard Bellman. 1966. Dynamic Programming. *Science* 153, 3731 (1966), 34–37. DOI:http://dx.doi.org/10.1126/science.153.3731.34

[19] Erin Beneteau, Olivia K. Richards, Mingrui Zhang, Julie A. Kientz, Jason Yip, and Alexis Hiniker. 2019. Communication Breakdowns Between Families and Alexa. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 243, 13 pages. DOI:http://dx.doi.org/10.1145/3290605.3300473

[20] Yoshua Bengio. 2009. *Learning deep architectures for AI*. Now Publishers Inc.

[21] Kathleen Benitez and Bradley Malin. 2010. Evaluating re-identification risks with respect to the HIPAA privacy rule. *Journal of the American Medical Informatics Association* 17, 2 (2010), 169–177.

[22] Frank Bentley, Chris Luvogt, Max Silverman, Rushani Wirasinghe, Brooke White, and Danielle Lottridge. 2018. Understanding the Long-Term Use of Smart Speaker Assistants. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 3, Article Article 91 (Sept. 2018), 24 pages. DOI:http://dx.doi.org/10.1145/3264901

[23] Frank R. Bentley, S. Tejaswi Peesapati, and Karen Church. 2016. "I Thought She Would Like to Read It": Exploring Sharing Behaviors in the Context of Declining Mobile Web Use. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1893–1903. DOI:http://dx.doi.org/10.1145/2858036.2858056

[24] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 1533–1544.

[25] Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. 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)*. ACM, New York, NY, USA, 191–200. DOI:http://dx.doi.org/10.1145/1095034.1095067

[26] Alan W. Biermann. 1983. Natural Language Programming. In *Computer Program Synthesis Methodologies (NATO Advanced Study Institutes Series)*, Alan W. Biermann and Gerard Guiho (Eds.). Springer Netherlands, 335–368.

[27] Jeffrey P. Bigham, Tessa Lau, and Jeffrey Nichols. 2009. Trailblazer: Enabling Blind Users to Blaze Trails through the Web. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 177–186. DOI:http://dx.doi.org/10.1145/1502650.1502677

[28] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. 2008. Robot programming by demonstration. In *Springer handbook of robotics*. Springer, 1371–1394. http://link.springer.com/10.1007/978-3-540-30301-5_60

[29] Michael Blackstock and Rodger Lea. 2014. IoT interoperability: A hub-based approach. In *Internet of Things (IOT), 2014 International Conference on the*. IEEE, 79–84. http://ieeexplore.ieee.org/abstract/document/7030119/

[30] Daniel G Bobrow, Ronald M Kaplan, Martin Kay, Donald A Norman, Henry Thompson, and Terry Winograd. 1977. GUS, a frame-driven dialog system. *Artificial intelligence* 8, 2 (1977), 155–173.

[31] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi. 2008. End-user programming in the wild: A field study of CoScripter scripts. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 39–46. DOI:http://dx.doi.org/10.1109/VLHCC.2008.4639056

[32] Dan Bohus and Alexander I. Rudnicky. 2005. Sorry, I didn't catch that!-An investigation of non-understanding errors and recovery strategies. In *6th SIGdial Workshop on Discourse and Dialogue*.

[33] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*. ACM, 163–172. http://dl.acm.org/citation.cfm?id=1095062

[34] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1247–1250. http://dl.acm.org/citation.cfm?id=1376746

[35] Richard A. Bolt. 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)*. ACM, New York, NY, USA, 262–270.

[36] Antoine Bosselut, Hannah Rashkin, Maarten Sap, Chaitanya Malaviya, Asli Celikyilmaz, and Yejin Choi. 2019. COMET: Commonsense Transformers for Automatic Knowledge Graph Construction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. ACL, Florence, Italy, 4762–4779. DOI:http://dx.doi.org/10.18653/v1/P19-1470

[37] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. ACL, Lisbon, Portugal, 632–642. DOI:http://dx.doi.org/10.18653/v1/D15-1075

[38] Susan E. Brennan. 1991. Conversation with and through computers. *User Modeling and User-Adapted Interaction* 1, 1 (01 Mar 1991), 67–86. DOI:http://dx.doi.org/10.1007/BF00158952

[39] Susan E Brennan. 1998. The grounding problem in conversations with and through computers. *Social and cognitive approaches to interpersonal communication* (1998), 201–225.

[40] Barry Brown, Moira McGregor, and Eric Laurier. 2013. iPhone in Vivo: Video Analysis of Mobile Device Use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1031–1040. DOI:http://dx.doi.org/10.1145/2470654.2466132

[41] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and others. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[42] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. 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)*. ACM, New York, NY, USA, 47–56. DOI:http://dx.doi.org/10.1145/2037373.2037383

[43] Juan Pablo Carrascal and Karen Church. 2015. An In-Situ Study of Mobile App & Mobile Search Interactions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 2739–2748. DOI:http://dx.doi.org/10.1145/2702123.2702486

[44] Pew Research Center. Communications Technology in Emerging and Developing Nations. (????). http://www.pewglobal.org/2015/03/19/1-communications-technology-in-emerging-and-developing-nations/

[45] Pew Research Center. 2019. Demographics of Mobile Device Ownership and Adoption in the United States. (2019). https://www.pewresearch.org/internet/fact-sheet/mobile/

[46] Vageesh Chandramouli, Abhijnan Chakraborty, Vishnu Navda, Saikat Guha, Venkata Padmanabhan, and Ramachandran Ramjee. 2015. Insider: Towards breaking down mobile app silos. In *TRIOS Workshop held in conjunction with the SIGOPS SOSP 2015*.

[47] Kerry Shih-Ping Chang, Brad A. Myers, Gene M. Cahill, Soumya Simanta, Edwin Morris, and Grace Lewis. 2013a. Improving Structured Data Entry on Mobile Devices. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 75–84. DOI:http://dx.doi.org/10.1145/2501988.2502043

[48] K. S. P. Chang, B. A. Myers, G. M. Cahill, S. Simanta, E. Morris, and G. Lewis. 2013b. A plug-in architecture for connecting to new data sources on mobile devices. In *2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 51–58. DOI:http://dx.doi.org/10.1109/VLHCC.2013.6645243

[49] Sarah Chasins and Rastislav Bodik. 2017. Skip blocks: reusing execution history to accelerate web scripts. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 51.

[50] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 963–975. DOI:http://dx.doi.org/10.1145/3242587.3242661

[51] Chen Chen, Simon T. Perrault, Shengdong Zhao, and Wei Tsang Ooi. 2014. BezelCopy: An Efficient Cross-application Copy-paste Technique for Touchscreen Smartphones. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces (AVI '14)*. ACM, New York, NY, USA, 185–192. DOI:http://dx.doi.org/10.1145/2598153.2598162

[52] Fanglin Chen, Kewei Xia, Karan Dhabalia, and Jason I. Hong. 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, Article Paper 575, 14 pages. DOI:http://dx.doi.org/10.1145/3290605.3300805

[53] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 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)*.

[54] Jiun-Hung Chen and Daniel S. Weld. 2008. Recovering from Errors During Programming by Demonstration. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '08)*. ACM, New York, NY, USA, 159–168. DOI:http://dx.doi.org/10.1145/1378773.1378794

[55] Merav Chkroun and Amos Azaria. 2019. LIA: A Virtual Assistant that Can Be Taught New Commands by Speech. *International Journal of Human–Computer Interaction* (2019), 1–12.

[56] Janghee Cho and Emilee Rader. 2020. The Role of Conversational Grounding in Supporting Symbiosis Between People and Digital Assistants. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1 (May 2020).

[57] Karen Church and Rodrigo de Oliveira. 2013. What's Up with Whatsapp?: Comparing Mobile Instant Messaging Behaviors with Traditional SMS. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services (MobileHCI '13)*. ACM, New York, NY, USA, 352–361. DOI:http://dx.doi.org/10.1145/2493190.2493225

[58] Karen Church, Denzil Ferreira, Nikola Banovic, and Kent Lyons. 2015. Understanding the Challenges of Mobile Phone Usage Data. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '15)*. ACM, New York, NY, USA, 504–514. DOI:http://dx.doi.org/10.1145/2785830.2785891

[59] Karen Church and Nuria Oliver. 2011. Understanding Mobile Web and Mobile Search Use in Today's Dynamic Mobile Landscape. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '11)*. ACM, New York, NY, USA, 67–76. DOI:http://dx.doi.org/10.1145/2037373.2037385

[60] Herbert H. Clark and Susan E. Brennan. 1991. Grounding in communication. In *Perspectives on socially shared cognition*. APA, Washington, DC, US, 127–149. DOI:http://dx.doi.org/10.1037/10096-006

[61] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[62] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis.. In *NDSS*.

[63] J. Coutaz and J. L. Crowley. 2016. A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing* 15, 2 (April 2016), 26–39. DOI:http://dx.doi.org/10.1109/MPRV.2016.24

[64] Benjamin R. Cowan, Nadia Pantidi, David Coyle, Kellie Morrissey, Peter Clarke, Sara Al-Shehri, David Earley, and Natasha Bandeira. 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)*. ACM, New York, NY, USA, Article 43, 12 pages. DOI:http://dx.doi.org/10.1145/3098279.3098539

[65] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.

[66] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 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)*. ACM, New York, NY, USA, 845–854. DOI:http://dx.doi.org/10.1145/3126594.3126651

[67] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 767–776. DOI:http://dx.doi.org/10.1145/2984511.2984581

[68] A. Demeure, S. Caffiau, E. Elias, and C. Roux. 2015. Building and Using Home Automation Systems: A Field Study. In *End-User Development (Lecture Notes in Computer Science)*, Paloma Díaz, Volkmar Pipek, Carmelo Ardito, Carlos Jensen, Ignacio Aedo, and Alexander Boden (Eds.). Springer International Publishing, 125–140. DOI:http://dx.doi.org/10.1007/978-3-319-18425-8_9

[69] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. ACL, Minneapolis, Minnesota, 4171–4186. DOI:http://dx.doi.org/10.18653/v1/N19-1423

[70] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. 2004. a CAPpella: programming by demonstration of context-aware applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 33–40. http://dl.acm.org/citation.cfm?id=985697

[71] Tilman Dingler and Martin Pielot. 2015. I'Ll Be There for You: Quantifying Attentiveness Towards Mobile Messaging. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '15)*. ACM, New York, NY, USA, 1–5. DOI:http://dx.doi.org/10.1145/2785830.2785840

[72] Morgan Dixon and James Fogarty. 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)*. ACM, New York, NY, USA, 1525–1534. DOI:http://dx.doi.org/10.1145/1753326.1753554

[73] Morgan Dixon, Daniel Leventhal, and James Fogarty. 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)*. ACM, New York, NY, USA, 969–978. DOI:http://dx.doi.org/10.1145/1978942.1979086

[74] Morgan Dixon, Alexander Nied, and James Fogarty. 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)*. ACM, New York, NY, USA, 221–230. DOI:http://dx.doi.org/10.1145/2642918.2647412

[75] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 225–234. DOI:http://dx.doi.org/10.1145/2047196.2047226

[76] W. Keith Edwards and Rebecca E. Grinter. 2001. At Home with Ubiquitous Computing: Seven Challenges. In *Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp '01)*. Springer-Verlag, London, UK, UK, 256–272. http://dl.acm.org/citation.cfm?id=647987.741327

[77] Khaled El Emam, Elizabeth Jonker, Luk Arbuckle, and Bradley Malin. 2011. A systematic review of re-identification attacks on health data. *PloS one* 6, 12 (2011), e28071.

[78] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[79] Sara Evensen, Chang Ge, and Cagatay Demiralp. 2020. Ruler: Data Programming by Demonstration for Document Labeling. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1996–2005. DOI:http://dx.doi.org/10.18653/v1/2020.findings-emnlp.181

[80] Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S. Bernstein. 2018. Iris: A Conversational Agent for Complex Tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 473:1–473:12. DOI:http://dx.doi.org/10.1145/3173574.3174047

[81] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. ACM, New York, NY, USA, 614–626. DOI:http://dx.doi.org/10.1145/3379337.3415869

[82] Earlence Fernandes, Oriana Riva, and Suman Nath. 2016. Appstract: On-the-fly App Content Semantics with Better Privacy. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking (MobiCom '16)*. ACM, New York, NY, USA, 361–374. DOI:http://dx.doi.org/10.1145/2973750.2973770

[83] Denzil Ferreira, Jorge Goncalves, Vassilis Kostakos, Louise Barkhuus, and Anind K. Dey. 2014. Contextual Experience Sampling of Mobile Application Micro-usage. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services (MobileHCI '14)*. ACM, New York, NY, USA, 91–100. DOI:http://dx.doi.org/10.1145/2628363.2628367

[84] A. Fox, B. Johanson, P. Hanrahan, and T. Winograd. 2000. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics and Applications* 20, 3 (May 2000), 54–65. DOI:http://dx.doi.org/10.1109/38.844373

[85] Martin R. Frank and James D. Foley. 1994. A Pure Reasoning Engine for Programming by Demonstration. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94)*. ACM, New York, NY, USA, 95–101. DOI:http://dx.doi.org/10.1145/192426.192466

[86] C. Ailie Fraser, Tricia J. Ngoon, Mira Dontcheva, and Scott Klemmer. 2019. RePlay: Contextually Presenting Learning Videos Across Software Applications. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 297, 13 pages. DOI:http://dx.doi.org/10.1145/3290605.3300527

[87] Vittorio Fuccella, Poika Isokoski, and Benoit Martin. 2013. Gestures and Widgets: Performance in Text Editing on Multi-touch Capable Mobile Devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 2785–2794. DOI:http://dx.doi.org/10.1145/2470654.2481385

[88] Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces*. ACM, 93–100.

[89] Kiev Gama, Lionel Touseau, and Didier Donsez. 2012. Combining heterogeneous service technologies for building an Internet of Things middleware. *Computer Communications* 35, 4 (2012), 405–417. http://www.sciencedirect.com/science/article/pii/S0140366411003586

[90] Michelle Gantt and Bonnie A. Nardi. 1992. Gardeners and Gurus: Patterns of Cooperation Among CAD Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*. ACM, New York, NY, USA, 107–117. DOI:http://dx.doi.org/10.1145/142750.142767

[91] Xiaofeng Gao, Ran Gong, Yizhou Zhao, Shu Wang, Tianmin Shu, and Song-Chun Zhu. 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)*. IEEE, 1119–1126.

[92] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of Context-Dependent Applications Through Trigger-Action Rules. *ACM Trans. Comput.-Hum. Interact.* 24, 2, Article 14 (April 2017), 33 pages. DOI:http://dx.doi.org/10.1145/3057861

[93] Kevin A Gluck and John E Laird. 2019. *Interactive Task Learning: Humans, Robots, and Agents Acquiring New Tasks through Natural Interactions*. Vol. 26. MIT Press.

[94] Cristian González García, B. Cristina Pelayo G-Bustelo, Jordán Pascual Espada, and Guillermo Cueva-Fernandez. 2014. Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios. *Computer Networks* 64 (May 2014), 143–158. DOI:http://dx.doi.org/10.1016/j.comnet.2014.02.010

[95] Google. 2016. Android Developer Reference - Activity. (2016). http://developer.android.com/reference/android/app/Activity.html

[96] Google. 2019. Advertising ID | Android Developers. (2019). http://www.androiddocs.com/google/play-services/id.html

[97] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. 1025–1035.

[98] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating Photo Manipulation Tutorials by Demonstration. In *ACM SIGGRAPH 2009 Papers (SIGGRAPH '09)*. ACM, New York, NY, USA, 66:1–66:9. DOI:http://dx.doi.org/10.1145/1576246.1531372

[99] Thomas RG Green. 1989. Cognitive dimensions of notations. *People and computers V* (1989), 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

[100] T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (June 1996), 131–174. DOI:http://dx.doi.org/10.1006/jvlc.1996.0009

[101] H Paul Grice, Peter Cole, Jerry Morgan, and others. 1975. Logic and conversation. *1975* (1975), 41–58.

[102] Jonathan Grudin and Richard Jacques. 2019. Chatbots, humbots, and the quest for artificial general intelligence. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–11.

[103] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. 2009. Towards physical mashups in the web of things. In *Networked Sensing Systems (INSS), 2009 Sixth International Conference on*. IEEE, 1–4. http://ieeexplore.ieee.org/abstract/document/5409925/

[104] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. DOI:http://dx.doi.org/10.1145/1926385.1926423

[105] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive Programming Meets the Real World. *Commun. ACM* 58, 11 (Oct. 2015), 90–99. DOI:http://dx.doi.org/10.1145/2736282

[106] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, and others. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[107] Anhong Guo, Junhan Kong, Michael Rivera, Frank F Xu, and Jeffrey P Bigham. 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)*. 15.

[108] Izzeddin Gur, Semih Yavuz, Yu Su, and Xifeng Yan. 2018. DialSQL: Dialogue Based Structured Query Generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. ACL, Melbourne, Australia, 1339–1349. DOI:http://dx.doi.org/10.18653/v1/P18-1124

[109] Daniel C. Halbert. 1993. SmallStar: programming by demonstration in the desktop metaphor. In *Watch what I do*. MIT Press, 103–123.

[110] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 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)*. ACM, New York, NY, USA, 241–250. DOI:http://dx.doi.org/10.1145/1294211.1294254

[111] Eric Horvitz. 1999. Principles of Mixed-Initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 159–166. DOI:http://dx.doi.org/10.1145/302979.303030

[112] Forrest Huang, John F. Canny, and Jeffrey Nichols. 2019. Swire: Sketch-Based User Interface Retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, 1–10. DOI:http://dx.doi.org/10.1145/3290605.3300334

[113] Justin Huang and Maya Cakmak. 2015. Supporting Mental Model Accuracy in Trigger-Action Programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 215–225. DOI:http://dx.doi.org/10.1145/2750858.2805830

[114] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 977–992. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/huang

[115] Ting-Hao Kenneth Huang, Amos Azaria, and Jeffrey P. Bigham. 2016. InstructableCrowd: Creating IF-THEN Rules via Conversations with the Crowd. ACM Press, 1555–1562. DOI:http://dx.doi.org/10.1145/2851581.2892502

[116] Connor Huff and Dustin Tingley. 2015. "Who are these people?" Evaluating the demographic characteristics and political preferences of MTurk survey respondents. *Research & Politics* 2, 3 (2015), 2053168015604648.

[117] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1986. Direct manipulation interfaces. (1986).

[118] Soshi Iba, Christiaan J. J. Paredis, and Pradeep K. Khosla. 2005. Interactive Multimodal Robot Programming. *The International Journal of Robotics Research* 24, 1 (Jan. 2005), 83–104. DOI:http://dx.doi.org/10.1177/0278364904049250

[119] IFTTT. 2016. IFTTT: connects the apps you love. (2016). https://ifttt.com/

[120] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. 2019. HILC: Domain-Independent PbD System Via Computer Vision and Follow-Up Questions. *ACM Trans. Interact. Intell. Syst.* 9, 2-3, Article 16 (March 2019), 27 pages. DOI:http://dx.doi.org/10.1145/3234508

[121] Mohit Jain, Pratyush Kumar, Ishita Bhansali, Q Vera Liao, Khai Truong, and Shwetak Patel. 2018a. FarmChat: A Conversational Agent to Answer Farmer Queries. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 170.

[122] Mohit Jain, Pratyush Kumar, Ramachandra Kota, and Shwetak N Patel. 2018b. Evaluating and informing the design of chatbots. In *Proceedings of the 2018 Designing Interactive Systems Conference*. ACM, 895–906.

[123] Chakajkla Jesdabodi and Walid Maalej. 2015. Understanding Usage States on Mobile Devices. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 1221–1225. DOI:http://dx.doi.org/10.1145/2750858.2805837

[124] Jiepu Jiang, Wei Jeng, and Daqing He. 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*. ACM, 143–152.

[125] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. 2018. Why Are They Collecting My Data? Inferring the Purposes of Network Traffic in Mobile Apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 1–27.

[126] Simon L. Jones, Denzil Ferreira, Simo Hosio, Jorge Goncalves, and Vassilis Kostakos. 2015. Revisitation Analysis of Smartphone App Use. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 1197–1208. DOI:http://dx.doi.org/10.1145/2750858.2807542

[127] Daniel Jurafsky and James H Martin. 2019. Dialogue Systems and Chatbots. *Speech and Language Processing* (2019).

[128] Amy K. Karlson, Shamsi T. Iqbal, Brian Meyers, Gonzalo Ramos, Kathy Lee, and John C. Tang. 2010. Mobile Taskflow in Context: A Screenshot Study of Smartphone Usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2009–2018. DOI:http://dx.doi.org/10.1145/1753326.1753631

[129] Amy K. Karlson, Brian R. Meyers, Andy Jacobs, Paul Johns, and Shaun K. Kane. 2009. Working overtime: Patterns of smartphone and PC usage in the day of an information worker. In *Pervasive computing*. Springer, 398–405. http://link.springer.com/chapter/10.1007/978-3-642-01516-8_27

[130] Tejaswi Kasturi, Haojian Jin, Aasish Pappu, Sungjin Lee, Beverley Harrison, Ramana Murthy, and Amanda Stent. 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*. 441–443.

[131] Artem Katasonov, Olena Kaykova, Oleksiy Khriyenko, Sergiy Nikitin, and Vagan Y. Terziyan. 2008. Smart Semantic Middleware for the Internet of Things. *ICINCO-ICSO* 8 (2008), 169–178. https://pdfs.semanticscholar.org/4665/130fd1236d7cf5636b188ca7adcc4fd8d631.pdf

[132] Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to Transform Natural to Formal Languages. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3 (AAAI'05)*. AAAI Press, Pittsburgh, Pennsylvania, 1062–1068. http://dl.acm.org/citation.cfm?id=1619499.1619504

[133] Simon Khalaf. Seven Years Into The Mobile Revolution: Content is King... Again. (????).

[134] Donghwi Kim, Sooyoung Park, Jihoon Ko, Steven Y. Ko, and Sung-Ju Lee. 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)*. ACM, New York, NY, USA, 95–108. DOI: http://dx.doi.org/10.1145/3332165.3347890

[135] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[136] James Kirk, Aaron Mininger, and John Laird. 2016. Learning task goals interactively with visual demonstrations. *Biologically Inspired Cognitive Architectures* 18 (2016), 1–8.

[137] James R. Kirk and John E. Laird. 2019. Learning Hierarchical Symbolic Representations to Support Interactive Task Learning and Knowledge Transfer. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. 6095–6102. DOI:http://dx.doi.org/10.24963/ijcai.2019/844

[138] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.* 43, 3 (April 2011), 21:1–21:44. DOI:http://dx.doi.org/10.1145/1922649.1922658

[139] Per Ola Kristensson, Stephen Brewster, James Clawson, Mark Dunlop, Leah Findlater, Poika Isokoski, Benoît Martin, Antti Oulasvirta, Keith Vertanen, and Annalu Waller. 2013. Grand Challenges in Text Entry. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems (CHI EA '13)*. ACM, New York, NY, USA, 3315–3318. DOI:http://dx.doi.org/10.1145/2468356.2479675

[140] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3083–3092. DOI:http://dx.doi.org/10.1145/2470654.2466420

[141] Kazutaka Kurihara, Masataka Goto, Jun Ogata, and Takeo Igarashi. 2006. Speech pen: predictive handwriting based on ambient multimodal recognition. In *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 851–860.

[142] Igor Labutov, Shashank Srivastava, and Tom Mitchell. 2018. LIA: A natural language programmable personal assistant. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 145–150.

[143] J. E. Laird, K. Gluck, J. Anderson, K. D. Forbus, O. C. Jenkins, C. Lebiere, D. Salvucci, M. Scheutz, A. Thomaz, G. Trafton, R. E. Wray, S. Mohan, and J. R. Kirk. 2017. Interactive Task Learning. *IEEE Intelligent Systems* 32, 4 (2017), 6–21. DOI:http://dx.doi.org/10.1109/MIS.2017.3121552

[144] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[145] Gierad P. Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. 2013. PixelTone: A Multimodal Interface for Image Editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 2185–2194. DOI:http://dx.doi.org/10.1145/2470654.2481301

[146] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (Oct. 2009), 65–67. http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262

[147] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2001. Your Wish is My Command. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Learning Repetitive Text-editing Procedures with SMARTedit, 209–226. http://dl.acm.org/citation.cfm?id=369505.369519

[148] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.* 53, 1-2 (Oct. 2003), 111–156. DOI:http://dx.doi.org/10.1023/A:1025671410623

[149] Chunggi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Young-Woo Park, Bum Chul Kwon, and Sungahn Ko. 2020a. 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)*. ACM, New York, NY, USA, 1–13. DOI:http://dx.doi.org/10.1145/3313831.3376327

[150] Hsin-Ying Lee, Weilong Yang, Lu Jiang, Madison Le, Irfan Essa, Haifeng Gong, and Ming-Hsuan Yang. 2020b. Neural Design Network: Graphic Layout Generation with Constraints. *European Conference on Computer Vision (ECCV)* (2020).

[151] Tak Yeon Lee, Casey Dugan, and Benjamin B. Bederson. 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)*. ACM, New York, NY, USA, 257–261. DOI:http://dx.doi.org/10.1145/3025171.3025203

[152] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1719–1728. DOI:http://dx.doi.org/10.1145/1357054.1357323

[153] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 73–84. DOI:http://dx.doi.org/10.14778/2735461.2735468

[154] Hao Li, Yu-Ping Wang, Jie Yin, and Gang Tan. 2019. SmartShell: Automated Shell Scripts Synthesis from Natural Language. *International Journal of Software Engineering and Knowledge Engineering* 29, 02 (2019), 197–220.

[155] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 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)*. ACM, New York, NY, USA, 723–732. DOI:http://dx.doi.org/10.1145/1753326.1753432

[156] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[157] Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. 2017. Adversarial Learning for Neural Dialogue Generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Copenhagen, Denmark, 2157–2169. DOI:http://dx.doi.org/10.18653/v1/D17-1230

[158] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. 2019. LayoutGAN: Synthesizing Graphic Layouts with Vector-Wireframe Adversarial Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).

[159] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA. DOI:http://dx.doi.org/10.1145/3025453.3025483

[160] Toby Jia-Jun Li, Jingya Chen, Brandon Canfield, and Brad A. Myers. 2020. Privacy-Preserving Script Sharing in GUI-Based Programming-by-Demonstration Systems. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1, Article 060 (May 2020), 23 pages. DOI:http://dx.doi.org/10.1145/3392869

[161] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M. Mitchell, and Brad A. Myers. 2020a. 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. DOI:http://dx.doi.org/10.1145/3379337.3415820

[162] Toby Jia-Jun Li and Brent Hecht. 2014. WikiBrain: Making Computer Programs Smarter with Knowledge from Wikipedia. (2014).

[163] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Tom M. Mitchell, and Brad A. Myers. 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)*.

[164] Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A. Myers. 2017. Programming IoT Devices by Demonstration Using Mobile Apps. In *End-User Development*, Simone Barbosa, Panos Markopoulos, Fabio Paterno, Simone Stumpf, and Stefano Valtolina (Eds.). Springer International Publishing, Cham, 3–17.

[165] Toby Jia-Jun Li, Tom Mitchell, and Brad Myers. 2020b. Interactive Task Learning from GUI-Grounded Natural Language Instructions and Demonstrations. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations (ACL 2020)*. ACL, 215–223. https://www.aclweb.org/anthology/2020.acl-demos.25

[166] Toby Jia-Jun Li, Lindsay Popowski, Tom M. Mitchell, and Brad A. Myers. 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.

[167] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 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. DOI:http://dx.doi.org/10.1145/3332165.3347899

[168] Toby Jia-Jun Li and Oriana Riva. 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.

[169] Toby Jia-Jun Li, Shilad Sen, and Brent Hecht. 2014. Leveraging Advances in Natural Language Processing to Better Understand Tobler's First Law of Geography. In *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '14)*. ACM, New York, NY, USA, 513–516. DOI:http://dx.doi.org/10.1145/2666310.2666493

[170] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2017. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 76 (Sept. 2017), 26 pages. DOI:http://dx.doi.org/10.1145/3130941

[171] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020b. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. ACL, Online, 8198–8210. DOI:http://dx.doi.org/10.18653/v1/2020.acl-main.729

[172] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020a. 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)*. ACL, Online, 5495–5510. DOI:http://dx.doi.org/10.18653/v1/2020.emnlp-main.443

[173] Percy Liang, Michael I. Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics* 39, 2 (2013), 389–446.

[174] Zhung-Xun Liao, Shou-Chung Li, Wen-Chih Peng, P.S. Yu, and Te-Chuan Liu. 2013. On the Feature Discovery for App Usage Prediction in Smartphones. In *2013 IEEE 13th International Conference on Data Mining (ICDM)*. 1127–1132. DOI:http://dx.doi.org/10.1109/ICDM.2013.130

[175] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

[176] Henry Lieberman and Hugo Liu. 2006. Feasibility studies for programming in natural language. In *End User Development*. Springer, 459–473.

[177] Henry Lieberman, Hugo Liu, Push Singh, and Barbara Barry. 2004. Beating Common Sense into Interactive Applications. *AI Magazine* 25, 4 (Dec. 2004), 63–63. DOI:http://dx.doi.org/10.1609/aimag.v25i4.1785

[178] H. Lieberman and D. Maulsby. 1996. Instructible agents: Software that just keeps getting better. *IBM Systems Journal* 35, 3.4 (1996), 539–556. DOI:http://dx.doi.org/10.1147/sj.353.0539

[179] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-User Development: An Emerging Paradigm. In *End User Development*. Springer, Dordrecht, 1–8. DOI:http://dx.doi.org/10.1007/1-4020-5386-X_1

[180] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI:http://dx.doi.org/10.1145/1502650.1502667

[181] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 2018b. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. *CoRR* abs/1802.08802 (2018). http://arxiv.org/abs/1802.08802

[182] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Taylor, Aniket Kittur, and Brad A. Myers. 2019. Unakite: Scaffolding Developers' Decision-Making Using the Web. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA, 67–80. DOI:http://dx.doi.org/10.1145/3332165.3347908

[183] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018a. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 569–579. DOI:http://dx.doi.org/10.1145/3242587.3242650

[184] LlamaLab. 2016. Automate: everyday automation for Android. (2016). http://llamalab.com/automate/

[185] Ewa Luger and Abigail Sellen. 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)*. ACM, New York, NY, USA, 5286–5297. DOI:http://dx.doi.org/10.1145/2858036.2858288

[186] Wendy E. Mackay. 1990. Patterns of Sharing Customizable Software. In *Proceedings of the 1990 ACM Conference on Computer-supported Cooperative Work (CSCW '90)*. ACM, New York, NY, USA, 209–221. DOI:http://dx.doi.org/10.1145/99332.99356

[187] I. Scott MacKenzie and R. William Soukoreff. 2002. Text entry for mobile computing: Models and methods, theory and practice. *Human–Computer Interaction* 17, 2-3 (2002), 147–198. http://www.tandfonline.com/doi/abs/10.1080/07370024.2002.9667313

[188] Christopher J. MacLellan, Erik Harpstead, Robert P. Marinier III, and Kenneth R. Koedinger. 2018. A Framework for Natural Cognitive System Training Interactions. *Advances in Cognitive Systems* (2018).

[189] Pattie Maes. 1994. Agents That Reduce Work and Information Overload. *Commun. ACM* 37, 7 (July 1994), 30–40. DOI:http://dx.doi.org/10.1145/176789.176792

[190] Jennifer Mankoff, Gregory D Abowd, and Scott E Hudson. 2000. OOPS: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers & Graphics* 24, 6 (2000), 819–834.

[191] Christoper Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52Nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. DOI:http://dx.doi.org/10.3115/v1/P14-5010

[192] R. Marin, P. J. Sanz, P. Nebot, and R. Wirz. 2005. A multimodal interface to control a robot arm via the web: a case study on remote programming. *IEEE Transactions on Industrial Electronics* 52, 6 (Dec. 2005), 1506–1520. DOI:http://dx.doi.org/10.1109/TIE.2005.858733

[193] Rodrigo de A. Maués and Simone Diniz Junqueira Barbosa. 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)*. ACM, New York, NY, USA, 295–303. DOI:http://dx.doi.org/10.1145/2493190.2493216

[194] Richard G. McDaniel and Brad A. Myers. 1997. Gamut: Demonstrating Whole Applications. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST '97)*. ACM, New York, NY, USA, 81–82. DOI:http://dx.doi.org/10.1145/263407.263515

[195] Richard G. McDaniel and Brad A. Myers. 1999. Getting More out of Programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 442–449. DOI:http://dx.doi.org/10.1145/302979.303127

[196] John H McDonald. 2009. *Handbook of Biological Statistics*. Vol. 2. Sparky House Publishing.

[197] Michael McTear, Ian O'Neill, Philip Hanna, and Xingkun Liu. 2005. Handling errors and determining confirmation strategies—An object-based approach. *Speech Communication* 45, 3 (2005), 249 – 269. DOI:http://dx.doi.org/10.1016/j.specom.2004.11.006 Special Issue on Error Handling in Spoken Dialogue Systems.

[198] Sarah Mennicken, Jo Vermeulen, and Elaine M. Huang. 2014. From Today's Augmented Houses to Tomorrow's Smart Homes: New Directions for Home Automation Research. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '14)*. ACM, New York, NY, USA, 105–115. DOI:http://dx.doi.org/10.1145/2632048.2636076

[199] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. 187–195. http://machinelearning.wustl.edu/mlpapers/papers/ICML2013_menon13

[200] Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (Natural Language Processing) for NLP (Natural Language Programming). In *Computational Linguistics and Intelligent Text Processing (Lecture Notes in Computer Science)*, Alexander Gelbukh (Ed.). Springer Berlin Heidelberg, 319–330.

[201] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]* (Jan. 2013). http://arxiv.org/abs/1301.3781 arXiv: 1301.3781.

[202] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality

[203] Robert C Miller and Brad A Myers. 2002. Multiple selections in smart text editing. In *Proceedings of the 7th international conference on Intelligent user interfaces*. ACM, 103–110.

[204] Francesmary Modugno and Brad A. Myers. 1994. Pursuit: Graphically Representing Programs in a Demonstrational Visual Shell. In *Conference Companion on Human Factors in Computing Systems (CHI '94)*. ACM, New York, NY, USA, 455–456. DOI:http://dx.doi.org/10.1145/259963.260464

[205] Shiwali Mohan and John E. Laird. 2014. Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI'14)*. AAAI Press, 387–394.

[206] Brad Myers, Robert Malkin, Michael Bett, Alex Waibel, Ben Bostwick, Robert C Miller, Jie Yang, Matthias Denecke, Edgar Seemann, Jie Zhu, and others. 2002. Flexi-modal and multi-machine user interfaces. In *Proceedings. Fourth IEEE International Conference on Multimodal Interfaces*. IEEE, 343–348.

[207] Brad. A. Myers. 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)*. ACM, New York, NY, USA, 59–66. DOI:http://dx.doi.org/10.1145/22627.22349

[208] Brad A. Myers. 1993. Peridot: creating user interfaces by demonstration. In *Watch what I do*. MIT Press, 125–153.

[209] Brad A. Myers. 1998. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press/Addison-Wesley Publishing Co., 534–541.

[210] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. DOI:http://dx.doi.org/10.1109/MC.2016.200

[211] Brad A. Myers, Amy J. Ko, Chris Scaffidi, Stephen Oney, YoungSeok Yoon, Kerry Chang, Mary Beth Kery, and Toby Jia-Jun Li. 2017. Making End User Development More Natural. In *New Perspectives in End-User Development*. Springer, Cham, 1–22. DOI:http://dx.doi.org/10.1007/978-3-319-60291-2_1

[212] Brad A. Myers and Richard McDaniel. 2001. Sometimes you need a little intelligence, sometimes you need a lot. *Your Wish is My Command: Programming by Example. San Francisco, CA: Morgan Kaufmann Publishers* (2001), 45–60. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.8085&rep=rep1&type=pdf

[213] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52. DOI:http://dx.doi.org/10.1145/1015864.1015888

[214] Chelsea Myers, Anushay Furqan, Jessica Nebolsky, Karina Caro, and Jichen Zhu. 2018. Patterns for how users overcome obstacles in voice user interfaces. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–7.

[215] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Omnipress, Madison, WI, USA, 807–814.

[216] Bonnie A Nardi. 1993. *A small matter of programming: perspectives on end user computing*. MIT press.

[217] Bonnie A. Nardi, James R. Miller, and David J. Wright. 1998. Collaborative, Programmable Intelligent Agents. *Commun. ACM* 41, 3 (March 1998), 96–104. DOI:http://dx.doi.org/10.1145/272287.272331

[218] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API Embedding for API Usages and Applications. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE, 438–449. DOI:http://dx.doi.org/10.1109/ICSE.2017.47

[219] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.

[220] Philippe Oechslin. 2003. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*. Springer, 617–630.

[221] Sharon Oviatt. 1999a. Mutual disambiguation of recognition errors in a multimodal architecture. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 576–583.

[222] Sharon Oviatt. 1999b. Ten Myths of Multimodal Interaction. *Commun. ACM* 42, 11 (Nov. 1999), 74–81. DOI:http://dx.doi.org/10.1145/319382.319398

[223] Sharon Oviatt and Philip Cohen. 2000. Perceptual user interfaces: multimodal interfaces that process what comes naturally. *Commun. ACM* 43, 3 (2000), 45–53.

[224] John F. Pane, Brad A. Myers, and others. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies* 54, 2 (2001), 237–264. http://www.sciencedirect.com/science/article/pii/S1071581900904105

[225] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '13)*. ACM, New York, NY, USA, 275–284. DOI:http://dx.doi.org/10.1145/2493432.2493490

[226] Panupong Pasupat, Tian-Shun Jiang, Evan Liu, Kelvin Guu, and Percy Liang. 2018. Mapping natural language commands to web elements. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. ACL, Brussels, Belgium, 4970–4976. DOI:http://dx.doi.org/10.18653/v1/D18-1540

[227] Panupong Pasupat and Percy Liang. 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.

[228] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*. ACL, Doha, Qatar, 1532–1543. DOI:http://dx.doi.org/10.3115/v1/D14-1162

[229] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers) (NAACL '18)*. ACL, New Orleans, Louisiana, 2227–2237. DOI:http://dx.doi.org/10.18653/v1/N18-1202

[230] Benny Pinkas and Tomas Sander. 2002. Securing passwords against dictionary attacks. In *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 161–170.

[231] Antonio Pintus, Davide Carboni, and Andrea Piras. 2011. The Anatomy of a Large Scale Social Web for Internet Enabled Objects. In *Proceedings of the Second International Workshop on Web of Things (WoT '11)*. ACM, New York, NY, USA, 6:1–6:6. DOI:http://dx.doi.org/10.1145/1993966.1993975

[232] Volkmar Pipek and Helge Kahler. 2006. Supporting Collaborative Tailoring. In *End User Development*, Henry Lieberman, Fabio Paternò, and Volker Wulf (Eds.). Springer Netherlands, Dordrecht, 315–345. DOI:http://dx.doi.org/10.1007/1-4020-5386-X_15

[233] Martin Porcheron, Joel E. Fischer, Stuart Reeves, and Sarah Sharples. 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, Article Paper 640, 12 pages. DOI:http://dx.doi.org/10.1145/3173574.3174214

[234] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces (IUI '00)*. ACM, New York, NY, USA, 207–211. DOI:http://dx.doi.org/10.1145/325737.325845

[235] Siyuan Qi, Baoxiong Jia, Siyuan Huang, Ping Wei, and Song-Chun Zhu. 2020. A Generalized Earley Parser for Human Activity Parsing and Prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[236] Philip Quinn and Shumin Zhai. 2016. A Cost-Benefit Study of Text Entry Suggestion Interaction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 83–88. DOI:http://dx.doi.org/10.1145/2858036.2858305

[237] Lenin Ravindranath, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. 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)*. ACM, New York, NY, USA, 4:1–4:6. DOI:http://dx.doi.org/10.1145/2162081.2162087

[238] Nils Reimers and Iryna Gurevych. 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

[239] V. Ricquebourg, D. Menga, D. Durand, B. Marhic, L. Delahoche, and C. Loge. 2006. The Smart Home Concept : our immediate future. In *2006 1ST IEEE International Conference on E-Learning in Industrial Electronics*. 23–28. DOI:http://dx.doi.org/10.1109/ICELIE.2006.347206

[240] André Rodrigues. 2015. Breaking Barriers with Assistive Macros. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility (ASSETS '15)*. ACM, New York, NY, USA, 351–352. DOI:http://dx.doi.org/10.1145/2700648.2811322

[241] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 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)*. ACM, New York, NY, USA, 275–284. DOI:http://dx.doi.org/10.1145/2494603.2480308

[242] Maarten Sap, Ronan Le Bras, Emily Allaway, Chandra Bhagavatula, Nicholas Lourie, Hannah Rashkin, Brendan Roof, Noah A Smith, and Yejin Choi. 2019. Atomic: An atlas of machine commonsense for if-then reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3027–3035.

[243] Ritam Sarmah, Yunpeng Ding, Di Wang, Cheuk Yin Phipson Lee, Toby Jia-Jun Li, and Xiang 'Anthony' Chen. 2020. Geno: a Developer Tool for Authoring Multimodal Interaction on Existing Web Applications. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST 2020)*.

[244] Albrecht Schmidt. 2015. Programming Ubiquitous Computing Environments. In *End-User Development (Lecture Notes in Computer Science)*, Paloma Díaz, Volkmar Pipek, Carmelo Ardito, Carlos Jensen, Ignacio Aedo, and Alexander Boden (Eds.). Springer International Publishing, 3–6. DOI:http://dx.doi.org/10.1007/978-3-319-18425-8_1

[245] Klaus Schwab. 2017. *The Fourth Industrial Revolution*. Crown Publishing Group, USA.

[246] Iulian V Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau. 2016. Building end-to-end dialogue systems using generative hierarchical neural network models. In *The 30th AAAI Conference on Artificial Intelligence (AAAI '16)*.

[247] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. 2020. VASTA: a vision and language-assisted smartphone task automation system. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*. 22–32.

[248] Lanbo She and Joyce Chai. 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)*. Association for Computational Linguistics, Vancouver, Canada, 1634–1644. DOI:http://dx.doi.org/10.18653/v1/P17-1150

[249] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. 2011. LiveLab: Measuring Wireless Networks and Smartphone Users in the Field. *SIGMETRICS Perform. Eval. Rev.* 38, 3 (Jan. 2011), 15–20. DOI:http://dx.doi.org/10.1145/1925019.1925023

[250] Hyo Sang Shin. 2012. *Visual Reading and The Snowball of Understanding* (1 edition ed.). Cheese Cake & Linco Global.

[251] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (Aug. 1983), 57–69. DOI:http://dx.doi.org/10.1109/MC.1983.1654471

[252] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. 2016. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (6 edition ed.). Pearson, Boston.

[253] Katie A. Siek, Gillian R. Hayes, Mark W. Newman, and John C. Tang. 2014. *Field Deployments: Knowing from Using in Context*. Springer New York, New York, NY, 119–142. DOI:http://dx.doi.org/10.1007/978-1-4939-0378-8_6

[254] Timothy Sohn, Kevin A. Li, William G. Griswold, and James D. Hollan. 2008. A Diary Study of Mobile Information Needs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 433–442. DOI:http://dx.doi.org/10.1145/1357054.1357125

[255] Z. Song, A. A. Cardenas, and R. Masuoka. 2010. Semantic middleware for the Internet of Things. In *2010 Internet of Things (IOT)*. 1–8. DOI:http://dx.doi.org/10.1109/IOT.2010.5678448

[256] Vijay Srinivasan, Saeed Moghaddam, Abhishek Mukherji, Kiran K. Rachuri, Chenren Xu, and Emmanuel Munguia Tapia. 2014. MobileMiner: Mining Your Frequent Patterns on Your Phone. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '14)*. ACM, New York, NY, USA, 389–400. DOI:http://dx.doi.org/10.1145/2632048.2632052

[257] Gaurav Srivastava, Saksham Chitkara, Kevin Ku, Swarup Kumar Sahoo, Matt Fredrikson, Jason I. Hong, and Yuvraj Agarwal. 2017a. PrivacyProxy: Leveraging Crowdsourcing and In Situ Traffic Analysis to Detect and Mitigate Information Leakage. *CoRR* abs/1708.06384 (2017). http://arxiv.org/abs/1708.06384

[258] Shashank Srivastava, Igor Labutov, and Tom Mitchell. 2017b. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1527–1536.

[259] Tom Stocky, Alexander Faaborg, and Henry Lieberman. 2004. A Commonsense Approach to Predictive Text Entry. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems (CHI EA '04)*. ACM, New York, NY, USA, 1163–1166. DOI:http://dx.doi.org/10.1145/985921.986014

[260] Anselm Strauss and Juliet M. Corbin. 1990. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc.

[261] Simone Stumpf, Vidya Rajaram, Lida Li, Margaret Burnett, Thomas Dietterich, Erin Sullivan, Russell Drummond, and Jonathan Herlocker. 2007. Toward Harnessing User Feedback for Machine Learning. In *Proceedings of the 12th International Conference on Intelligent User Interfaces (IUI '07)*. ACM, New York, NY, USA, 82–91. DOI:http://dx.doi.org/10.1145/1216295.1216316

[262] Yu Su, Ahmed Hassan Awadallah, Miaosen Wang, and Ryen W. White. 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)*. ACM, New York, NY, USA, 855–864. DOI:http://dx.doi.org/10.1145/3209978.3210013

[263] Bernhard Suhm, Brad Myers, and Alex Waibel. 2001. Multimodal Error Correction for Speech User Interfaces. *ACM Trans. Comput.-Hum. Interact.* 8, 1 (March 2001), 60–98. DOI:http://dx.doi.org/10.1145/371127.371166

[264] Ming Sun, Yun-Nung Chen, and Alexander I. Rudnicky. 2015. Understanding User's Cross-Domain Intentions in Spoken Dialog Systems. In *NIPS Workshop on Machine Learning for SLU and Interaction*.

[265] Ming Sun, Yun-Nung Chen, and Alexander I. Rudnicky. 2016a. HELPR: A Framework to Break the Barrier across Domains in Spoken Dialog Systems. In *International Workshop on Spoken Dialog Systems*. https://www.researchgate.net/profile/Yun_Nung_Chen/publication/291075015_HELPR_A_Framework_to_Break_the_Barrier_across_Domains_in_Spoken_Dialog_Systems/links/569de28408ae00e5c98f0aec.pdf

[266] Ming Sun, Yun-Nung Chen, and Alexander I. Rudnicky. 2016b. An Intelligent Assistant for High-Level Task Understanding. In *Proceedings of the 21st International Conference on Intelligent User Interfaces (IUI '16)*. ACM, New York, NY, USA, 169–174. DOI:http://dx.doi.org/10.1145/2856767.2856818

[267] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1501–1510. DOI:http://dx.doi.org/10.1145/3038912.3052709

[268] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J. Ko. 2018. Rewire: Interface Design Assistance from Examples. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 1–12. DOI:http://dx.doi.org/10.1145/3173574.3174078

[269] Ahmad Bisher Tarakji, Jian Xu, Juan A. Colmenares, and Iqbal Mohomed. 2018. Voice Enabling Mobile Applications with UIVoice. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking (EdgeSys'18)*. ACM, New York, NY, USA, 49–54. DOI:http://dx.doi.org/10.1145/3213344.3213353

[270] Jaime Teevan, Amy Karlson, Shahriyar Amini, A. J. Bernheim Brush, and John Krumm. 2011. Understanding the Importance of Location, Time, and People in Mobile Local Search Behavior. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '11)*. ACM, New York, NY, USA, 77–80. DOI:http://dx.doi.org/10.1145/2037373.2037386

[271] Michael Toomim, Steven M. Drucker, Mira Dontcheva, Ali Rahimi, Blake Thomson, and James A. Landay. 2009. Attaching UI Enhancements to Websites with End Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1859–1868. DOI:http://dx.doi.org/10.1145/1518701.1518987

[272] Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. Word Representations: A Simple and General Method for Semi-Supervised Learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL '10)*. ACL, USA, 384–394.

[273] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 803–812. DOI:http://dx.doi.org/10.1145/2556288.2557420

[274] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 3227–3231. DOI:http://dx.doi.org/10.1145/2858036.2858556

[275] David Vadas and James R Curran. 2005. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*. 191–199.

[276] Priyan Vaithilingam and Philip J Guo. 2019. Bespoke: Interactively synthesizing custom GUIs from command-line applications by demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST*, Vol. 19.

[277] Niels van Berkel, Chu Luo, Theodoros Anagnostopoulos, Denzil Ferreira, Jorge Goncalves, Simo Hosio, and Vassilis Kostakos. 2016. A Systematic Assessment of Smartphone Usage Gaps. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM.

[278] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. 2013. A taxonomy of privacy-preserving record linkage techniques. *Information Systems* 38, 6 (2013), 946–969.

[279] Keith Vertanen and Per Ola Kristensson. 2011. A Versatile Dataset for Text Entry Evaluations Based on Genuine Mobile Emails. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '11)*. ACM, New York, NY, USA, 295–298. DOI:http://dx.doi.org/10.1145/2037373.2037418

[280] Keith Vertanen, Per Ola Kristensson, Mark Dunlop, Ahmad Sabbir Arif, and James Clawson. 2016. Invisid Text Entry and Beyond. In *CHI '16 Extended Abstracts on Human Factors in Computing Systems*.

[281] Keith Vertanen, Haythem Memmi, Justin Emge, Shyam Reyal, and Per Ola Kristensson. 2015. VelociTap: Investigating Fast Mobile Text Entry Using Sentence-Based Decoding of Touchscreen Keyboard Input. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 659–668. DOI:http://dx.doi.org/10.1145/2702123.2702135

[282] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. http://dl.acm.org/citation.cfm?id=2629489

[283] Geraldine P Wallach and Katharine G Butler. 1994. *Language learning disabilities in school-age children and adolescents: Some principles and applications*. Allyn & Bacon.

[284] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1631–1634. DOI:http://dx.doi.org/10.1145/3035918.3058738

[285] Xiaolei Wang, Andrea Continella, Yuexiang Yang, Yongzhong He, and Sencun Zhu. 2019. LeakDoctor: Toward Automatically Diagnosing Privacy Leaks in Mobile Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 1, Article 28 (March 2019), 25 pages. DOI:http://dx.doi.org/10.1145/3314415

[286] Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. ACL, New Orleans, Louisiana, 1112–1122. DOI:http://dx.doi.org/10.18653/v1/N18-1101

[287] Kristin Williams. 2020. The IoT Codex: A Book of Paper Engineering Techniques for Authoring and Composing Embedded Computing Applications. In *Proceedings of the 11th Annual Workshop on the Intersection of HCI and PL (PLATEAU '20)*.

[288] Jacob O. Wobbrock and Brad A. Myers. 2006. Analyzing the Input Stream for Character- Level Errors in Unconstrained Text Entry Evaluations. *ACM Trans. Comput.-Hum. Interact.* 13, 4 (Dec. 2006), 458–489. DOI: http://dx.doi.org/10.1145/1188816.1188819

[289] Jacob O. Wobbrock, Brad A. Myers, and John A. Kembel. 2003. EdgeWrite: A Stylus-based Text Entry Method Designed for High Accuracy and Stability of Motion. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. ACM, New York, NY, USA, 61–70. DOI:http://dx.doi.org/10.1145/964696.964703

[290] Yu Wu, Wei Wu, Chen Xing, Can Xu, Zhoujun Li, and Ming Zhou. 2019. A Sequential Matching Framework for Multi-Turn Response Selection in Retrieval-Based Chatbots. *Computational Linguistics* 45, 1 (March 2019), 163–197. DOI:http://dx.doi.org/10.1162/coli_a_00345

[291] Volker Wulf. 1999. "Let's See Your Search-tool!"—Collaborative Use of Tailored Artifacts in Groupware. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP '99)*. ACM, New York, NY, USA, 50–59. DOI:http://dx.doi.org/10.1145/320297.320303

[292] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Michael L Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2017. Toward human parity in conversational speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25, 12 (2017), 2410–2423.

[293] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. 2011. Identifying Diverse Usage Behaviors of Smartphone Apps. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, New York, NY, USA, 329–344. DOI: http://dx.doi.org/10.1145/2068816.2068847

[294] Ye Xu, Mu Lin, Hong Lu, Giuseppe Cardone, Nicholas Lane, Zhenyu Chen, Andrew Campbell, and Tanzeem Choudhury. 2013. Preference, Context and Communities: A Multi-faceted Approach to Predicting Smartphone App Usage Patterns. In *Proceedings of the 2013 International Symposium on Wearable Computers (ISWC '13)*. ACM, New York, NY, USA, 69–76. DOI:http://dx.doi.org/10.1145/2493988.2494333

[295] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. ACM, New York, NY, USA, 113–126. DOI:http://dx.doi.org/10.1145/2307636.2307648

[296] Jackie (Junrui) Yang, Monica S. Lam, and James A. Landay. 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)*. ACM, New York, NY, USA, 35–44. DOI:http://dx.doi.org/10.1145/3379337.3415841

[297] Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 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)*. Association for Computational Linguistics, Hong Kong, China, 5447–5458. DOI:http://dx.doi.org/10.18653/v1/D19-1547

[298] Ziyu Yao, Yiqi Tang, Wen-tau Yih, Huan Sun, and Yu Su. 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)*. Association for Computational Linguistics, Online, 6883–6902. DOI:http://dx.doi.org/10.18653/v1/2020.emnlp-main.559

[299] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 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)*. ACM, New York, NY, USA, 183–192. DOI:http://dx.doi.org/10.1145/1622176.1622213

[300] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 476–486.

[301] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. *CoRR* abs/1704.01696 (2017). http://arxiv.org/abs/1704.01696

[302] Shumin Zhai, Michael Hunter, and Barton A. Smith. 2002. Performance Optimization of Virtual Keyboards. *Human–Computer Interaction* 17, 2-3 (Sept. 2002), 229–269. DOI:http://dx.doi.org/10.1080/07370024.2002.9667315

[303] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. ACM, New York, NY, USA, 627–648. DOI:http://dx.doi.org/10.1145/3379337.3415900

[304] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. 2017. Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6024–6037. DOI:http://dx.doi.org/10.1145/3025453.3025846

[305] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 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)*.

[306] Zhenliang Zhang, Yixin Zhu, and Song-Chun Zhu. 2020. Graph-based Hierarchical Knowledge Representation for Robot Task Transfer from Virtual to Physical World. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020).

[307] Sha Zhao, Julian Ramos, Jianrong Tao, Ziwen Jiang, Shijian Li, Zhaohui Wu, Gang Pan, and Anind K. Dey. 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)*. ACM, New York, NY, USA, 498–509. DOI:http://dx.doi.org/10.1145/2971648.2971696

[308] Yu Zhong, Yue Suo, Wenchang Xu, Chun Yu, Xinwei Guo, Yuhang Zhao, and Yuanchun Shi. 2011. Smart Home on Smart Phone. In *Proceedings of the 13th International Conference on Ubiquitous Computing (UbiComp '11)*. ACM, New York, NY, USA, 467–468. DOI:http://dx.doi.org/10.1145/2030112.2030174

[309] Xun Zou, Wangsheng Zhang, Shijian Li, and Gang Pan. 2013. Prophet: What App You Wish to Use Next. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication (UbiComp '13 Adjunct)*. ACM, New York, NY, USA, 167–170. DOI:http://dx.doi.org/10.1145/2494091.2494146