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

Ph.D. Thesis Proposal

Toby Jia-Jun Li
Human-Computer Interaction Institute
Carnegie Mellon University

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

November 26, 2019

Abstract

Intelligent agents that can perform tasks on behalf of users have become increasingly popular with growing ubiquity in "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 their 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 have designed SUGILITE, a new intelligent agent that allows end users to teach new tasks and concepts. 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 preliminary 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 and the relevant concepts for determining these conditions. The users also considered the multi-modal interaction style in SUGILITE natural and easy to use.

Over the course of this research, we have also developed three extensions to the original SUGILITE system: APPINITE, EPIDOSITE, and PUMICE. Through them we present *(i)* an new approach to allow the agent to generalize from learned task procedures by inferring task parameters and their associated possible values from user 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 agent conceptual knowledge used in conditionals, and *(iv)* a new mechanism to extend mobile app based PBD to smart home and Internet of Things (IoT) automation.

To complete the dissertation, I will explore several practical issues pertinent to the wide-adoption of the SUGILITE's approach for real-life usage. First, we will make the system more robust by enhancing its natural language understanding capability, and designing a more effective technique for end users to handle various types of errors that they may encounter when using PBD agents. Second, we plan to investigate privacy issues in sharing PBD scripts, and develop a new privacy-preserving mechanism for sharing the scripts. Lastly, enabled by the two previously proposed works, we will deploy SUGILITE with a group of actual users, and study how they use SUGILITE in real-life scenarios through an in-situ field study.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the widespread popularity of mobile apps, users are using them to complete a wide variety of tasks ranging from information query (e.g., checking weather), making purchases (e.g., ordering coffee), communicating (e.g., sending messages), to performing professional job duties (e.g., managing server configurations) [22, 159]. 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 familiar to users, which help users easily navigate around GUI structures to locate the desired functionalities [2, 34, 137].

However, GUI-based mobile apps have some limitations. First, performing tasks on GUIs can be tedious, especially when the task is repetitive. For example, the current version of the Starbucks app on Android requires 14 taps to order a cup of venti Iced Cappuccino with skim milk, and even more if the user does not have the account information stored. For such tasks, users would often like to have them automated [5, 107, 134]. Second, direct manipulation of GUIs is often not feasible or convenient in some contexts. Third, many tasks require coordination among many apps. But nowadays, data often remain siloed in individual apps [23]. In a formative study we did, 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. Lastly,

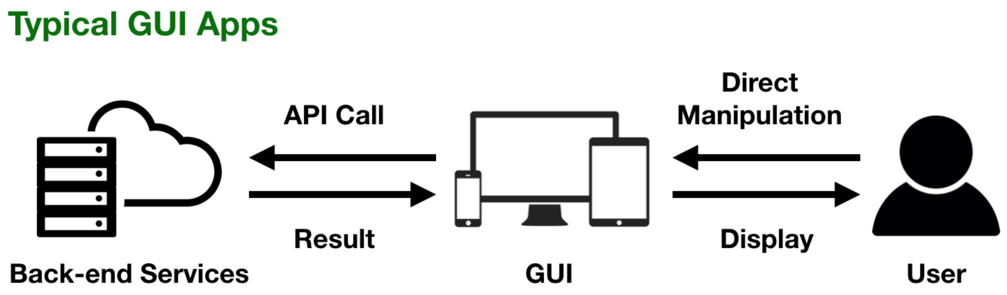


Figure 1.1: The typical role of a GUI. The GUI sits between the user and the back end services, where it receives direct manipulation inputs from the user, sends corresponding calls to the back-end services, and displays the results to the user.

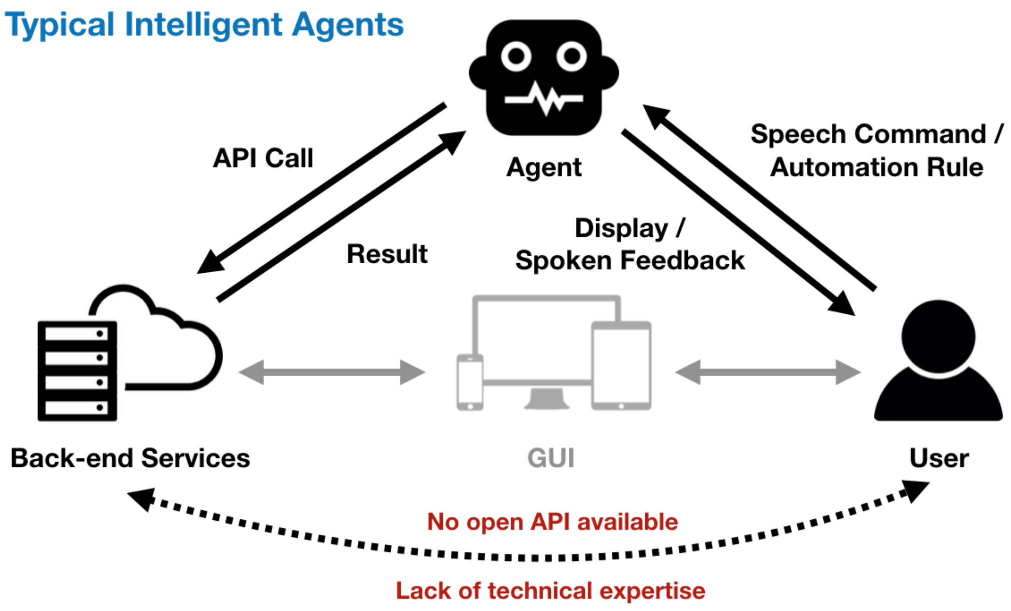

Typical Intelligent Agents
Agent
API Call
Result
Speech Command / Automation Rule
Display / Spoken Feedback
Back-end Services
GUI
User
No open API available
Lack of technical expertise

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.

while some app GUIs provide certain mechanisms of personalization (e.g., remembering and pre-filling the user's home location), they are mostly hard-coded. Users have few means of creating customized rules and specifying personalized task parameters to reflect their preferences beyond what the app developers have explicitly designed for.

Recently, intelligent agents 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 [103]. 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 [21, 139]. 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 better support hand-free contexts when the user is physically away from the device, cognitively occupied by other tasks (e.g., driving), or on devices with little or no screen space (e.g., wearables) [100]. 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 developers of third-party apps can integrate their apps into the agents to allow the agents to invoke these apps from user commands. However, such integration requires significant cost and engineering effort from app developers, therefore only some of the most popular tasks in popular apps have been integrated into prevailing intelligent agents so far. The "long-tail" of tasks and apps have not been supported yet, and will likely not get supported due to the cost and effort involved.

Prior literature [154] showed that the usage of "long-tail" apps made up significant portion in user app usage. Smartphone users also have highly diverse usage patterns within apps [159] and wish to have more customizability over how agents perform their tasks [32]. Therefore, relying on third-party developers' effort to extend the capabilities of intelligent agents is not sufficient for supporting diverse user needs. It is not feasible for end users to develop for new tasks in prevailing agents on their own either, due to *(i)* their lack of technical expertise required, and *(ii)* the limited availability of openly accessible application programming interfaces (APIs) for many back-end services, as shown in Figure 1.2.

1.1 An End-User Programmable Intelligent Agent

To address this problem, I propose to design, implement, and study a new end-user programmable intelligent agent called SUGILITE[1] [85] in this dissertation. The thesis for my dissertation is:

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 below key research challenges that SUGILITE seeks to address:

- **Usability:** SUGILITE should be usable for users without significant programming expertise. Some prior EUD systems (e.g., [99, 134]) 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., [13,27]), or services that provide open API access to their functionalities (e.g., [62]). 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 [136] 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.

- **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 [62,65], while providing great usability due to its simplicity, is not sufficiently expressive for many tasks that users want to automate [148].

[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.

- **Robustness:** SUGILITE should be resilient to minor changes in target applications, and be able to recover from errors caused by previously unseen or unexpected situations with the user's help. Macro recorders such as [136] 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., [98, 112]) usually lack transparency into the inference process, making it difficult for end users to recover from errors.

- **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 [82, 84], 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 the above challenges, SUGILITE takes a *multi-modal* approach, where it learns new tasks and concepts from end users 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.

By combining PBD and natural language programming, SUGILITE mitigates the crucial 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. It is also infeasible to have a computer system understand arbitrary user instructions in natural language without constraining it to specific domains and entities with current natural language processing (NLP) techniques. In my approach, SUGILITE *grounds* natural language instructions to demonstration app GUIs, allowing *mutual disambiguation* [124], 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.

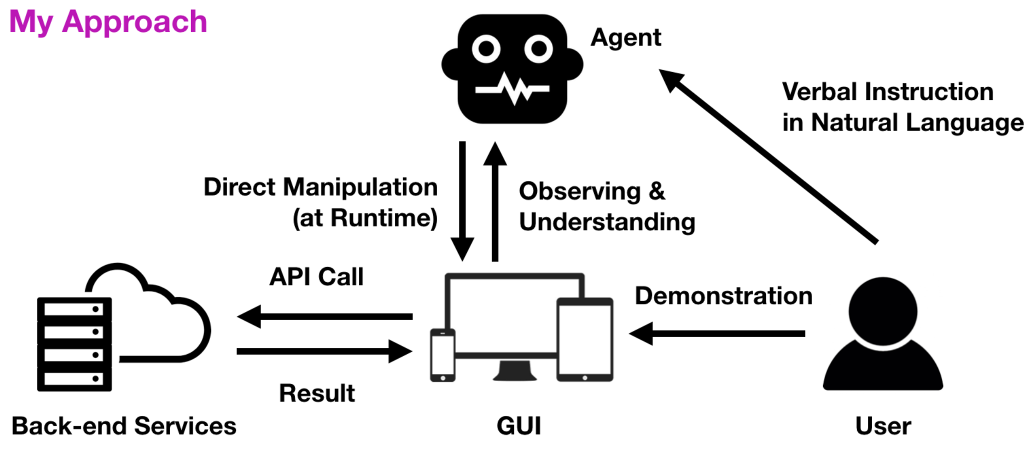

My Approach
Agent
Verbal Instruction in Natural Language
Direct Manipulation (at Runtime)
Observing & Understanding
API Call
Result
Demonstration
Back-end Services
GUI
User

Figure 1.3: The role of SUGILITE *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 direct manipulates the app GUI on the user's behalf to perform the task.

1.2 Contributions

My proposed 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, the preliminary work made the following contributions thus far:

- The prototype system of SUGILITE, a mobile PBD system that enables end users with no significant programming expertise to create automation scripts for arbitary tasks across any or multiple third-party mobile apps through a multi-modal interface combining demonstrations and natural language instructions [85] (Chapter 3).

- 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 [85] (Chapter 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 [86] (Chapter 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 [86] (Chapter 4).

- 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 [89] (Chapter 5).

- An extension of SUGILITE that leverages PBD on mobile apps to control smart home devices *through* their corresponding mobile app GUIs, addressing the *interoperability* challenge in smart home automation by allowing end users to create automation rules across multiple IoT devices from different manufactures or "eco-systems" [88] (Chapter 6).
- 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.

In the proposed work, I plan to make the following contributions:

- An error handling approach for programmable intelligent agents that can help users recover from common errors in various stages of the processing pipeline (i.e., speech recognition, intent classification, entity resolution, and task fulfillment).
- 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 the original scripts.
- An in-situ field study for the SUGILITE system with real users to understand its usage characteristics, and to evaluate its *usefulness* in real-life scenarios.

1.3 Outline

The following Chapter 2 will describe the background and related work. Chapter 3 will cover the design, implementation, and usability evaluation of the core SUGILITE system and its approach in further detail. Chapter 4 will discuss 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 will introduce another SUGILITE's extension named PUMICE that allows SUGILITE to learn conceptual knowledge in addition to procedural knowledge about tasks. Chapter 6 will talk about the application of SUGILITE's approach in smart home scenarios. Lastly, Chapter 7 proposes plans for improving SUGILITE's robustness through enhancing its natural language understanding capability and developing new error handling mechanisms, supporting privacy-preserving script sharing among users, and deploying SUGILITE in the field for an in-situ study.

Chapter 2

Background

This chapter discusses prior work on *(i)* programming by demonstration, *(ii)* natural language programming, *(iii)* multi-modal interfaces, *(iv)* end user development for task automation, and *(v)* understanding app interfaces.

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 [33, 93, 117].

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 [27] only worked with interfaces generated with the SUPPLE framework [48], and DocWizards [13] could only record actions performed on SWT widgets in Eclipse. Some PBD system focused on automating a specific task domain like file manipulation [115], photo manipulation [52], text editing [79], web tasks [10, 82], web scraping [25]), or generating interactive interfaces [46, 108, 151]. Much work has also been done on PBD for human-robot interaction (e.g., [6, 15, 73]).

SUGILITE supports domain-independent PBD by using GUIs of third-party apps. Similar approaches have also been used in prior systems. For example, Assistive Macros [136] used mobile app GUIs, CoScripter [82], d.mix [61], Vegemite [97], Ringer [10], and PLOW [3] used web interfaces, and HILC [67] and Sikuli [155] used desktop GUIs. Macro recording tools like [136] can record a sequence of input events and replay them later. These tools are too literal — they can only replay exactly the same procedure that was demonstrated, without the ability to generalize the demonstration to perform similar tasks. They are also brittle to any UI changes in the app. Sikuli [155] and HILC [66] 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 [155] require users to define conditionals in a scripting language, which is not feasible for end users without significant programming expertise.

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 [8,14, 94,113]. In NLP and AI communities, this approach is also known as learning by instruction [7,28,76,96].

The foremost challenge in supporting natural language programming is to deal with the inherent ambiguities and vagueness in natural language [150]. To address this challenge, one prior approach was to require users to use similar expression styles that resembled conventional programming languages (e.g., [8,83,133]), 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 [44] asked follow-up questions and presents possible options through conversations when initial user inputs are incomplete or unclear. This approach lowered the learning barrier for end users, as it did not require them to clearly define everything up front. It also allowed users to form complex commands by combining multiple natural language instructions in conversational turns under the guidance of the system. 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 [44], email processing [7, 143], or database queries [72].

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 on the specific task domain. This approach enables SUGILITE to support a wide range of tasks from diverse domains, as long as the corresponding mobile apps are available. This approach also has a low learning barrier because end users are already familiar with the functionalities of mobile apps and how to use them. In comparison, with prior instructable agents, users are often unclear about what concepts, procedures and functionalities already exist to be used as "building blocks" for developing new ones.

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 [125]. 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 [19]. Prior systems such as CommandSpace [1], Speechify [70], QuickSet [126], SMARTBoard [116], and PixelTone [77] 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., [64, 106]). But the use of these systems are limited to specific first-party apps and task domains, in contrast to SUGILITE which aims to be general-purpose.

When SUGILITE addresses the data description problem (details in Chapter 4), demonstration is the primary modality; verbal instructions are used for disambiguating demonstrated actions. A key pattern here is *mutual disambiguation* [124]. 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 users intent would be fundamentally unreliable. Similarly, from verbal instructions alone, the system may learn about the users 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 [104], such as correcting speech recognition errors [145] and assisting the recognition of pen-based handwriting [75].

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

Throughout the design process for SUGILITE, we used user-centered methods [120] including the *natural programming* method [121, 123]. 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 [53] to make it easier for developers to implement what they have in mind and to understand the state of their program [121]. 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., [62, 65, 149]) 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 [99] and AppGate [31] allowed users to create automation scripts with blocks in flowcharts. Helena [24] also used 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.

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 [38, 39, 40] introduced pixel-based methods to model interactive widgets and interface hierarchies in GUIs, and allowed runtime modifications of widget behaviors. Waken [9] also used a computer vision approach to recognize GUI components and activities from screen captured videos. RICO [34] and ERICA [35] showed that higher-level semantically meaningful tasks (e.g., search, login) can be recognized from GUI layouts of individual screens based on their design patterns. StateLens [59] and KITE [90] looked 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.

SUGILITE faces a unique challenge — in SUGILITE, the user talks about the underlying task of an app in natural language while making references to the app's GUI. The system needs to have sufficient understanding about the 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 understanding the user's instruction about the underlying task.

Chapter 3

The SUGILITE System

3.1 Introduction

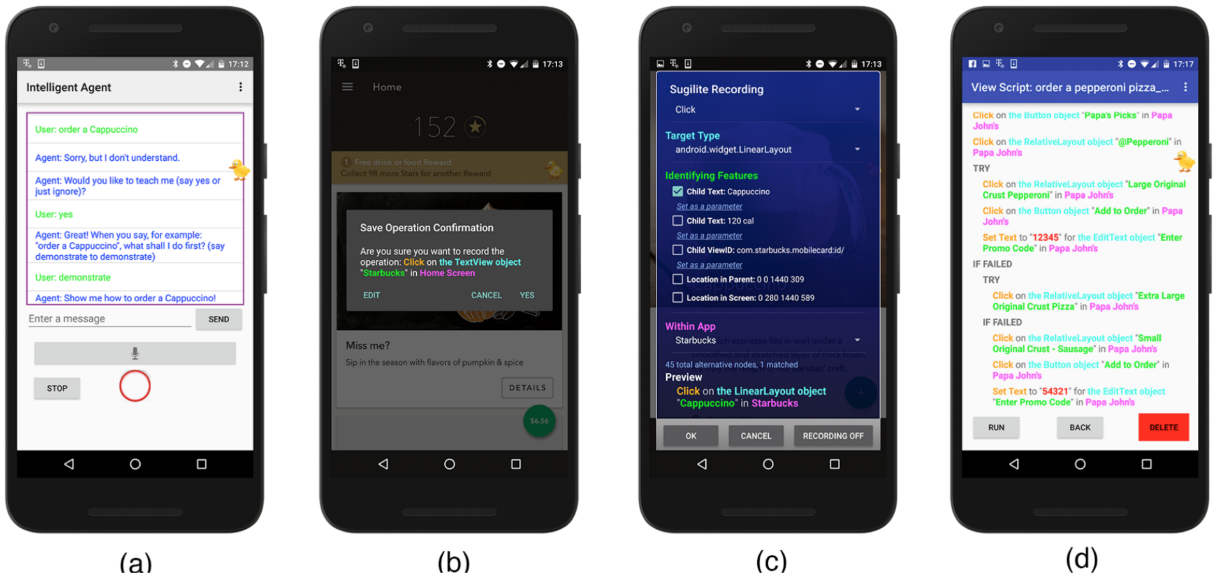
This chapter describes the design and implementation of the core SUGILITE system, a new intelligent agent that allows end users to teach new tasks and concepts using a *multi-modal* approach that combines *programming by demonstration* (PBD) and learning from natural language instructions. It features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

This chapter is modified from the conference paper [85].



Intelligent Agent

User: order a Cappuccino

Agent: Sorry, but I don't understand.

Agent: Would you like to teach me (say yes or just ignore)?

User: yes

Agent: Great! When you say, for example: "order a Cappuccino", what shall I do first? (say demonstrate to demonstrate)

User: demonstrate

Agent: Show me how to order a Cappuccino!

Enter a message

Home

152

1 Free drink or food Reward
Collect 98 more Stars for another Reward

Save Operation Confirmation

Are you sure you want to record the operation: Click on the TextView object "Starbucks" in Home Screen

Miss me?
Sip in the season with flavors of pumpkin & spice

$6.56

Sugilite Recording
Click

Target Type
android.widget.LinearLayout

Identifying Features
- [✓] Child Text: Cappuccino
  Set as a parameter
- [ ] Child Text: 120 cal
  Set as a parameter
- [ ] Child ViewID: com.starbucks.mobilecard:id/
  Set as a parameter
- [ ] Location in Parent: 0 0 1440 309
- [ ] Location in Screen: 0 280 1440 589

Within App
Starbucks

45 total alternative nodes, 1 matched

Preview
Click on the LinearLayout object "Cappuccino" in Starbucks

View Script: order a pepperoni pizza_...

```
Click on the Button object "Papa's Picks" in Papa John's
Click on the RelativeLayout object "@Pepperoni" in Papa John's
TRY
    Click on the RelativeLayout object "Large Original Crust Pepperoni" in Papa John's
    Click on the Button object "Add to Order" in Papa John's
    Set Text to "12345" for the EditText object "Enter Promo Code" in Papa John's
IF FAILED
    TRY
        Click on the RelativeLayout object "Extra Large Original Crust Pizza" in Papa John's
    IF FAILED
        Click on the RelativeLayout object "Small Original Crust - Sausage" in Papa John's
        Click on the Button object "Add to Order" in Papa John's
        Set Text to "54321" for the EditText object "Enter Promo Code" in Papa John's
```

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 the latest version, the conversational interface (a) has been replaced with* PUMICE *(Chapter 5), and the data description disambiguation interface (c) has been replaced with* APPINITE *(see Chapter 4).*

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

[1] A demo is available at https://www.youtube.com/watch?v=KMx7Ea6W6AQ

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 Starbucks "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. 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 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 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 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 plan to address this problem in the future, as discussed in Section 7.3.

3.3.3 Error Checking and Handling

Error checking and handling has been a major challenge for many PBD systems [78]. 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. For future work, we also plan to address other kinds of errors such as speech recognition errors, intent recognition errors, and entity resolution errors (see Section 7.2).

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 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 SUGILITE components and how they are integrated with each other. SUGILITE is an Android application implemented in Java. Scripts and tracking data are stored locally on phone in an SQLite database. SUGILITE does not require jailbreaking/root access to the phone and should work on any phone running Android 4.4 or above.

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] [7]. 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 [7].

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 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

[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.

(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, The 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.

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 [91] and the Yahoo InMind[5] Middleware have been 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.

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

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

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

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 pizzas (pepperoni, cheese and sausage). There were one `SET_TEXT` and 9 `CLICK` operations in the script. Among them, two `CLICK` operations required manual disambiguating 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`

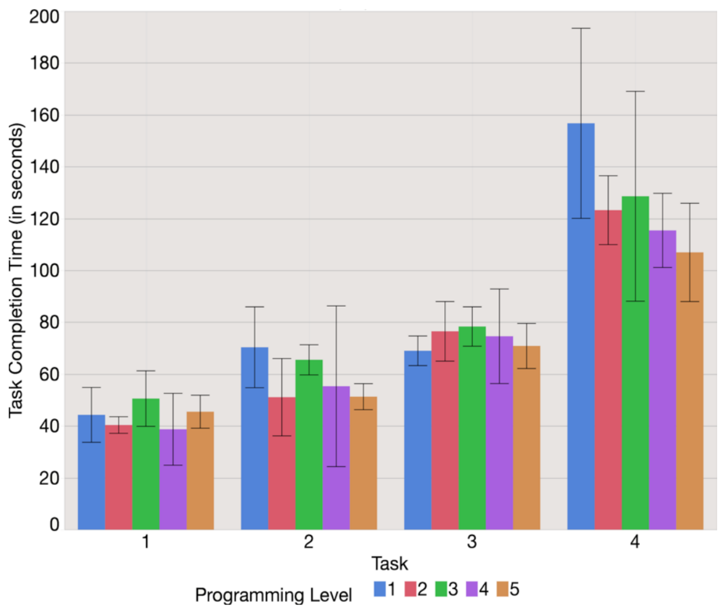


Figure 3.2: The average task completion time for the participants grouped by their programming experience. Shorter bars are better. Error bars show the standard deviations.

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 ($X^2(4) = 4.15$, $p = 0.39$).

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.

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).

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 participant 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* [114]. 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 for 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.

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

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].

[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.

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.

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 for 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 in 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 [85].

Chapter 4

The Data Description Problem

4.1 Introduction

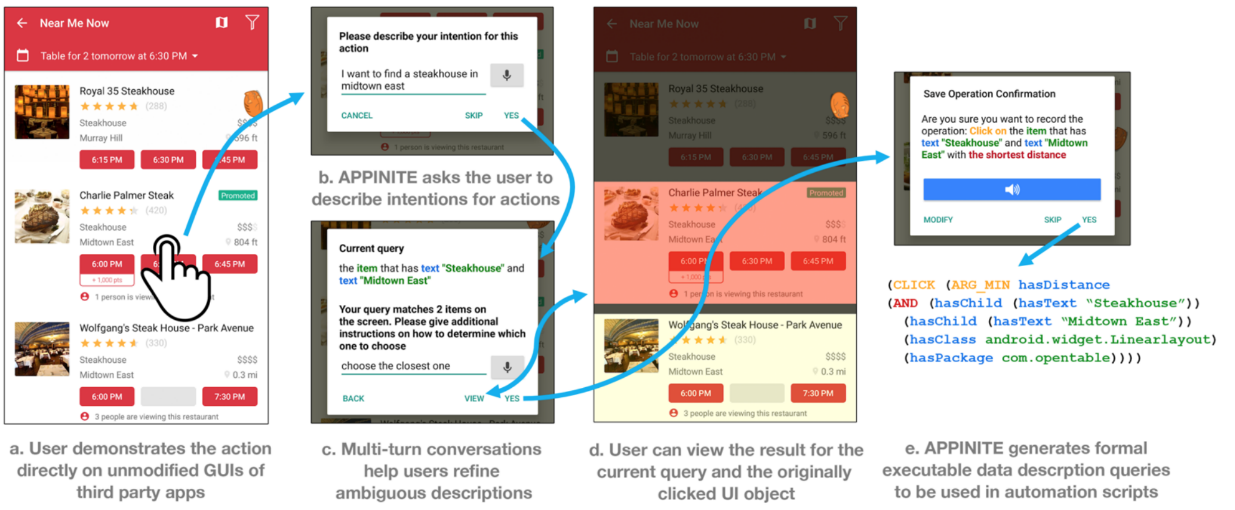
A central challenge for PBD is generalization. The PBD system should produce more than literal record-and-replay macros (e.g., sequences of keystrokes and clicks at specific screen locations), but learn the target task at a higher level of abstraction so it can perform similar tasks in new contexts [33, 93]. 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* [33, 93]: 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 [122]. Some prior systems such as SmallStar [60] and Topaz [119] used the "no inference" approach to give 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 [118] and CoScipter [82] 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 [58, 80, 109, 112] 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 [78]. Even if end users provide a sufficient number of examples, prior studies [81,

This chapter is modified from the conference paper [86].


Near Me Now
Table for 2 tomorrow at 6:30 PM
Royal 35 Steakhouse
Charlie Palmer Steak
Wolfgang's Steak House - Park Avenue
a. User demonstrates the action directly on unmodified GUIs of third party apps
b. APPINITE asks the user to describe intentions for actions
c. Multi-turn conversations help users refine ambiguous descriptions
d. User can view the result for the current query and the originally clicked UI object
e. APPINITE generates formal executable data descrption queries to be used in automation scripts
CANCEL
SKIP
YES
BACK
VIEW
MODIFY

Please describe your intention for this action

I want to find a steakhouse in midtown east

Current query

the item that has text "Steakhouse" and text "Midtown East"

Your query matches 2 items on the screen. Please give additional instructions on how to determine which one to choose

choose the closest one

Save Operation Confirmation

Are you sure you want to record the operation: Click on the item that has text "Steakhouse" and text "Midtown East" with the shortest distance

```
(CLICK (ARG_MIN hasDistance
(AND (hasChild (hasText "Steakhouse"))
  (hasChild (hasText "Midtown East"))
  (hasClass android.widget.Linearlayout)
  (hasPackage com.opentable))))
```

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.

122] 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 [78], 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.

[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.

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 [63]) 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 intentions 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).

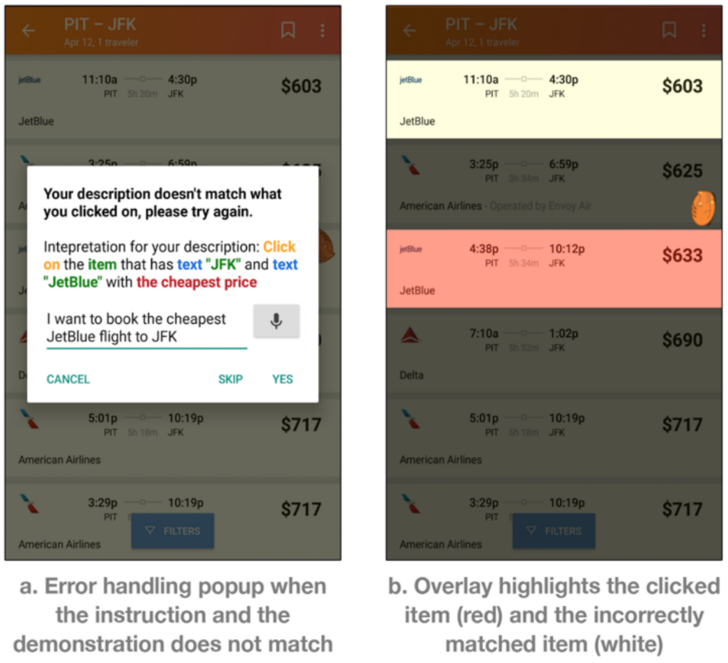
4.3 APPINITE's Design and Implementation

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

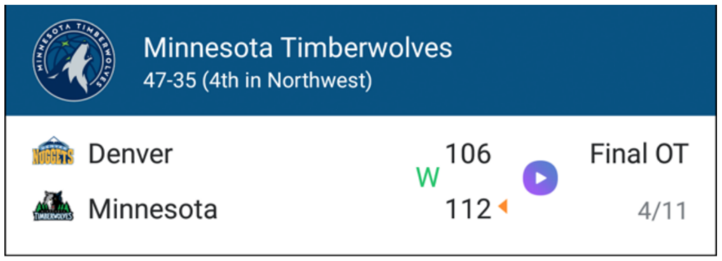

Your description doesn't match what you clicked on, please try again.

Intepretation for your description: Click on the item that has text "JFK" and text "JetBlue" with the cheapest price

I want to book the cheapest JetBlue flight to JFK

PIT – JFK
Apr 12, 1 traveler
11:10a 4:30p PIT 5h 20m JFK $603 JetBlue
3:25p 6:59p PIT 3h 34m JFK $625 American Airlines · Operated by Envoy Air
4:38p 10:12p PIT 5h 34m JFK $633 JetBlue
7:10a 1:02p PIT 5h 52m JFK $690 Delta
5:01p 10:19p PIT 5h 18m JFK $717 American Airlines
3:29p 10:19p PIT $717 American Airlines
FILTERS
CANCEL
SKIP
YES
a. Error handling popup when the instruction and the demonstration does not match
b. Overlay highlights the clicked item (red) and the incorrectly matched item (white)

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



```
(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."

ttree 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.

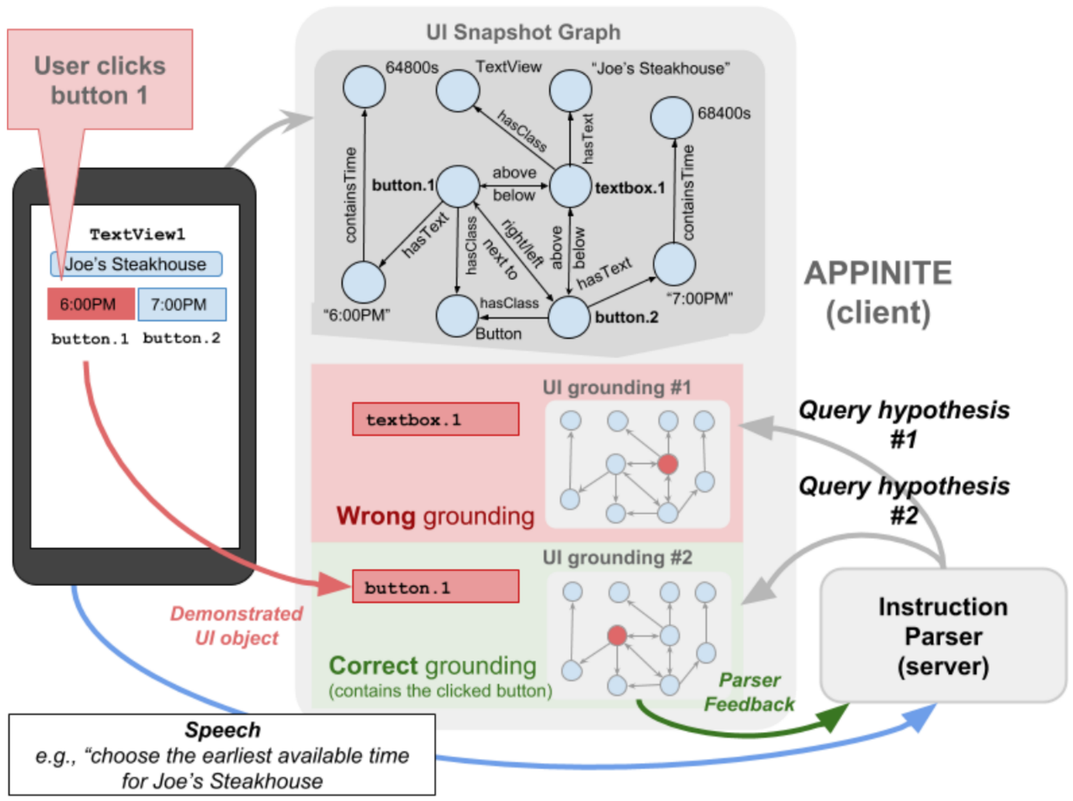
To represent and to execute queries used in data descriptions, APPINITE constructs relational knowledge graphs (i.e., ontologies) from hierarchical GUI structures as the medium-level representations for GUIs. These *UI snapshot graphs* abstract the semantics (values and relations) of GUIs from their 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 [42], Freebase [43], Wikidata [44] and WikiBrain [45], which can enable us to plug APPINITE's UI snapshot graph into these knowledge bases to support better semantic understanding of app GUIs in the future.

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

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



```
(CLICK (ARG_MIN numeric_time (AND
  (hasClass Button)
  (below (hasText "Joe's Steakhouse")))))
```

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

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 APPINITE'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:

$$E \to e; E \to S; S \to (\texttt{join}\ r\ E); S \to (\texttt{and}\ S\ S)$$

$$T \to (\texttt{ARG\_MAX}\ r\ S); T \to (\texttt{ARG\_MIN}\ r\ S); Q \to S \mid T$$

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 [92] a notationally simpler alternative to lambda calculus which is particularly well-suited for expressing queries over knowledge-graphs.

APPINITE's parser uses a Floating Parser architecture [129] 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 [12].

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 [18, 41, 147]). 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 problem 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 [158] 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 woman and 5 men, 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 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 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 (see Section 7.1). 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, σ = 0.4), and "easy to use" (mean = 7). They also agreed that "the multi-modal interface of APPINITE is helpful" (mean = 6.8, σ = 0.4), "the real-time visualization is helpful for disambiguation" (mean = 6.7, σ = 0.5), and "the error messages are helpful" (mean = 6.8, σ = 0.4).

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

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. As the next step, We plan to conduct an in-situ field study to evaluate APPINITE's usage and performance for organic tasks in real-world settings (see Section 7.4).

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 [128].

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.

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 [85] 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.

This chapter is modified from the conference paper [89].

[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

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 (NLP) 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 [102]. 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 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 [120] for designing a natural end-user development system [121]. 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 which are non-programmers; 38 men, 19 women, 1 non-binary person).

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 [120] 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 [144]. The inter-rater agreement [29] was $\kappa = 0.87$, suggesting good agreement. 19% of responses were excluded from the analysis for quality control due to the lack of efforts 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.

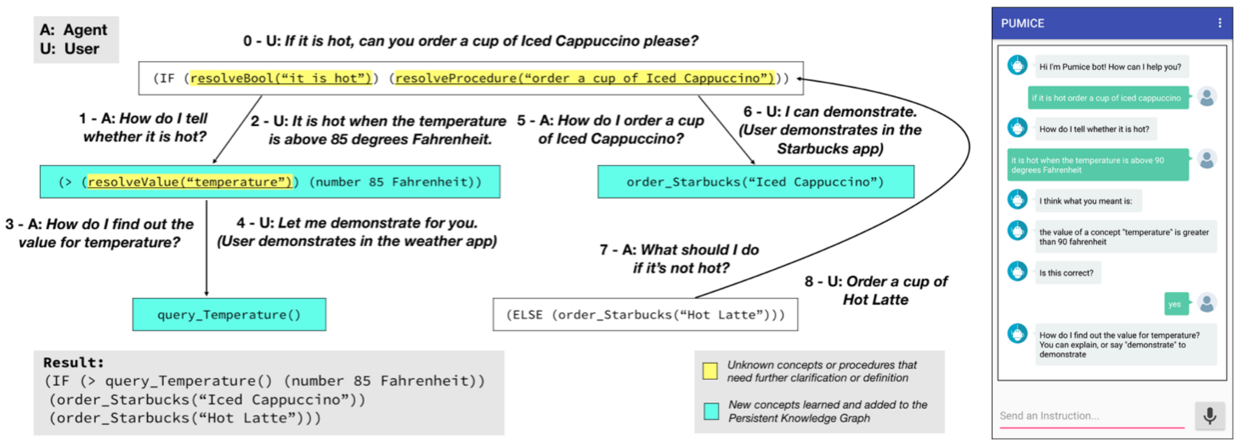
5.2.2 Unmet User Expectation of Common Sense Reasoning

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 expectation and communicating the agent's capability is also a long-standing unsolved challenge in building interactive intelligent systems [95]. 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.

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.


A: Agent
U: User
0 - U: If it is hot, can you order a cup of Iced Cappuccino please?
1 - A: How do I tell whether it is hot?
2 - U: It is hot when the temperature is above 85 degrees Fahrenheit.
3 - A: How do I find out the value for temperature?
4 - U: Let me demonstrate for you. (User demonstrates in the weather app)
5 - A: How do I order a cup of Iced Cappuccino?
6 - U: I can demonstrate. (User demonstrates in the Starbucks app)
7 - A: What should I do if it's not hot?
8 - U: Order a cup of Hot Latte
Unknown concepts or procedures that need further clarification or definition
New concepts learned and added to the Persistent Knowledge Graph
PUMICE
Send an Instruction...

```
(IF (resolveBool("it is hot")) (resolveProcedure("order a cup of Iced Cappuccino")))
(> (resolveValue("temperature")) (number 85 Fahrenheit))
order_Starbucks("Iced Cappuccino")
query_Temperature()
(ELSE (order_Starbucks("Hot Latte")))
```

```
Result:
(IF (> query_Temperature() (number 85 Fahrenheit))
 (order_Starbucks("Iced Cappuccino"))
 (order_Starbucks("Hot Latte")))
```

Hi I'm Pumice bot! How can I help you?

if it is hot order a cup of iced cappuccino

How do I tell whether it is hot?

it is hot when the temperature is above 90 degrees Fahrenheit

I think what you meant is:

the value of a concept "temperature" is greater than 90 fahrenheit

Is this correct?

yes

How do I find out the value for temperature? You can explain, or say "demonstrate" to demonstrate

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".

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 [54].

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.

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 user utterances using typed `resolve...()` functions, as marked in the yellow highlights in the parse for Utterance 0 in Figure 5.1. It then proceeds

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

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 5.2 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 [90]). To learn how to perform this action, it asks, "*How do I order a cup of Iced Cappuccino?*", to which the user responds, "*I 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 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 `or-der_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 results of the formative study.

Support for Concept Learning

In the formative study, we identified two main challenges with regard to concept learning. First, user 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), in this section, we focus on discussing how PUMICE learns Boolean concepts and value concepts.

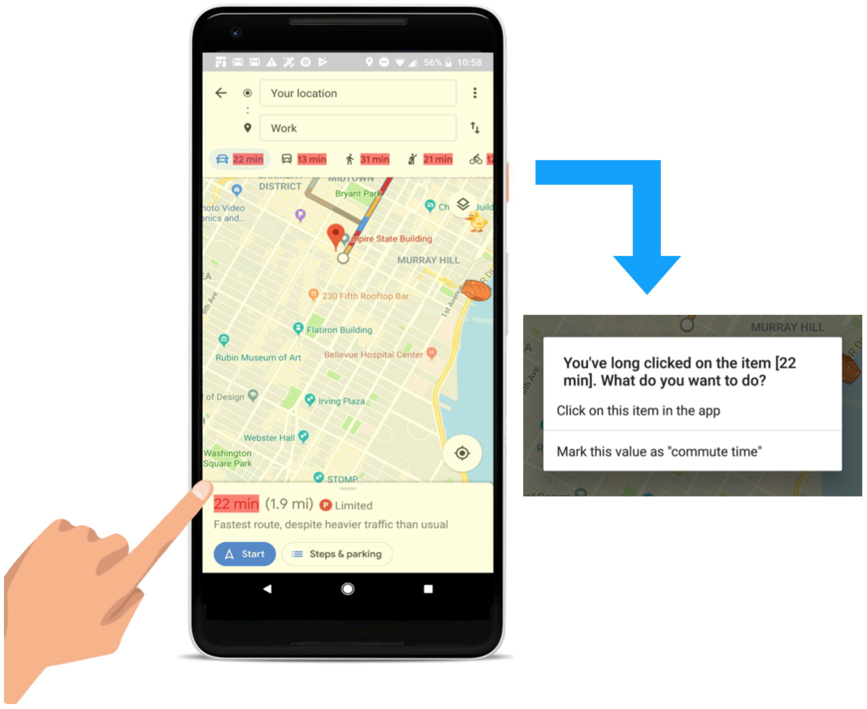
When encountering an unknown or unclear concept in the utterance parsing result, PUMICE first determines the concept type based on the context. If the concept is used as a condition (e.g., "if **it is hot**"), then it should be of Boolean type. Similarly, if a concept is used where a value is expected (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 5.2). 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 [158] 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,



You've long clicked on the item [22 min]. What do you want to do?

Click on this item in the app

Mark this value as "commute time"

Figure 5.2: 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.

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).

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 5.1).

For future work, we also plan to address other kinds of errors such as speech recognition errors, intent recognition errors, and entity resolution errors (see Section 7.2).

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 [12] 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 [129]. This approach also provides more interpretable results and requires less training data than neural network approaches (e.g., [156,157]). The parser parses user utterances into expressions in a simple functional DSL we created for PUMICE.

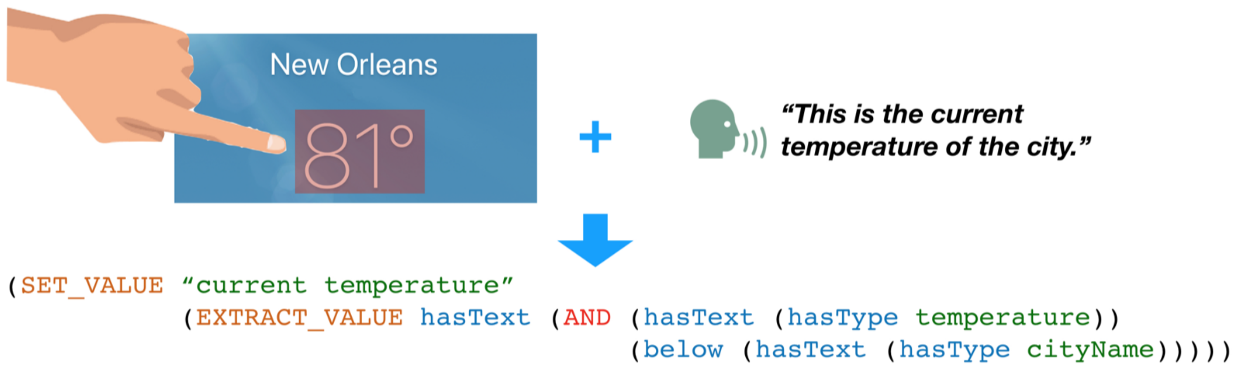
A key feature we added to PUMICE's parser is allowing typed `resolve()` functions in the parsing results to indicate unknown or unclear concepts and procedures. This feature adds interactivity to the traditional semantic parsing process. When this `resolve()` function is called at runtime, the front-end 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 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 [105] 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 5.3). 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.



```
(SET_VALUE "current temperature"
           (EXTRACT_VALUE hasText (AND (hasText (hasType temperature))
                                       (below (hasText (hasType cityName)))))
```

Figure 5.3: 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."

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 women, 5 men, 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

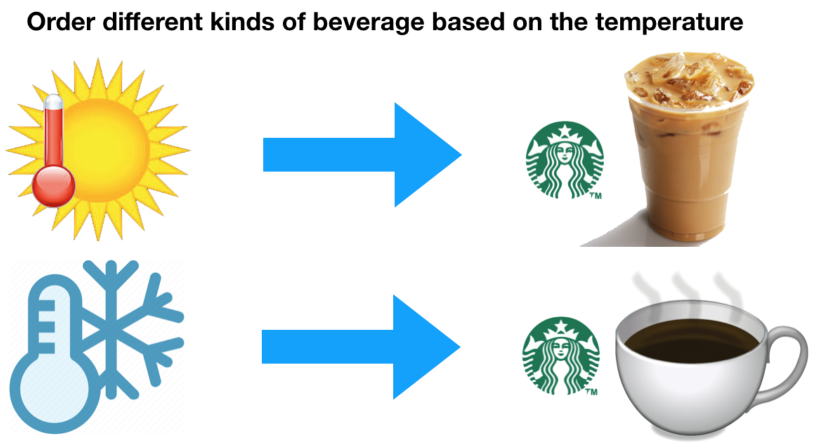


Figure 5.4: The graphical prompt used for Task 1 – A possible user command can be "*Order Iced coffee when it's hot outside, otherwise order hot coffee when the weather is cold.*"

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 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 [120]. Task descriptions were provided in the form of graphics, with minimal text descriptions that could not be directly used in user instructions (see Figure 5.4 for an example).

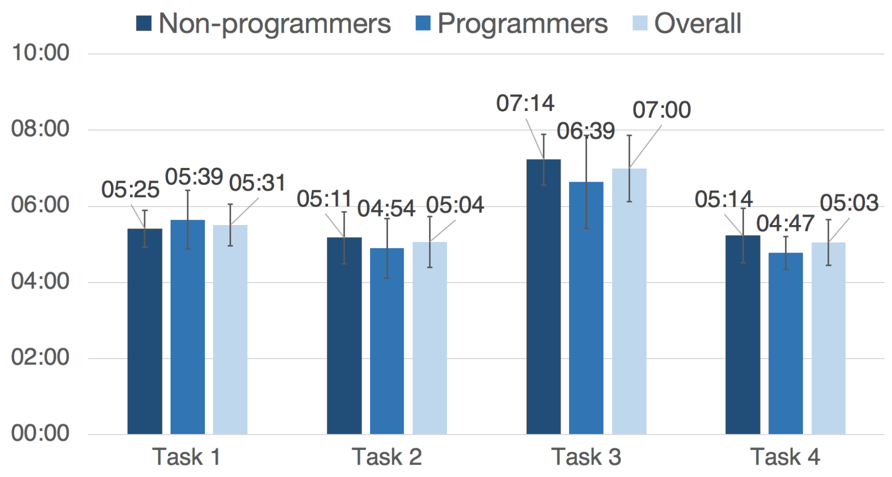


Figure 5.5: The average task completion times for each task. The error bars show one standard deviation in each direction.

Task 1 In this task, the user instructs PUMICE to order iced coffee when the weather is hot, and order hot coffee otherwise (Figure 5.4). 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 5.5 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 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 successes 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

Application in Smart Home Scenarios

6.1 Introduction

In the 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 [135]. 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 [148]. 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 [111, 138].

Many manufacturers of smart devices have provided their customers with tools for creating their own automations. 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 an automation to adjust the setting of an LG air conditioner based on the reading from a Samsung sensor.

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 automations 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

This chapter is modified from the conference paper [88].

[1] https://ifttt.com/

protocol, many existing tools and systems cannot support the heterogeneity of IoT devices [50,55]. While generalized architectures for programming IoT devices and higher-level representations of IoT automation rules and scripts have been proposed (e.g., [45,49,56,57,131]), 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 automations for IoT devices from different ecosystems by demonstration though 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 [160]. 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

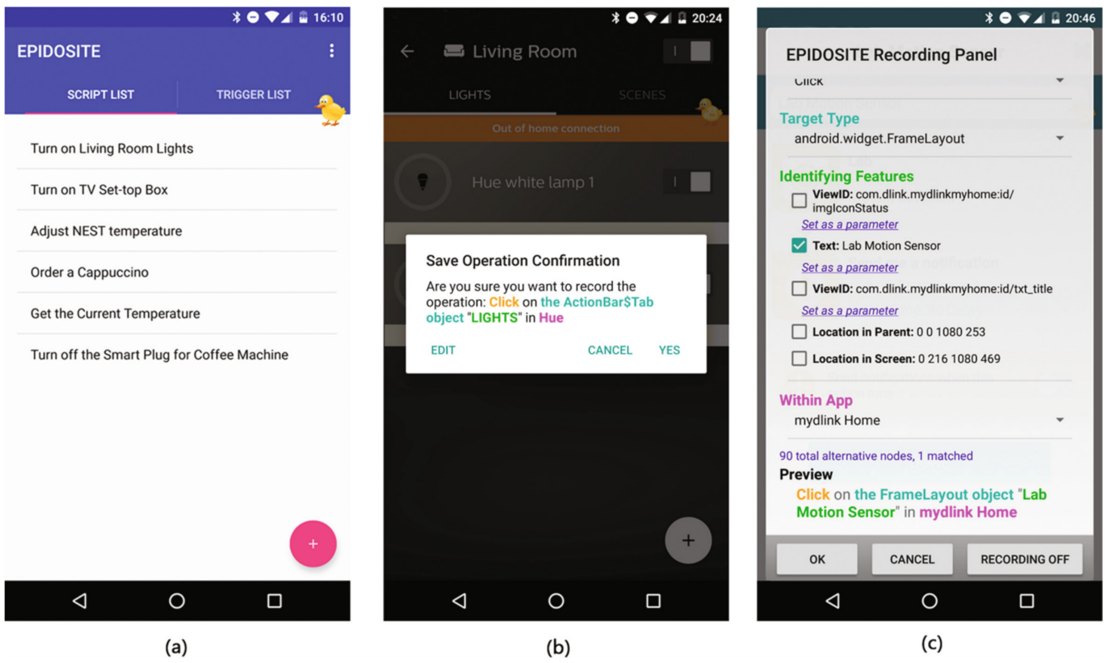
EPIDOSITE's approach has the following three major advantages:

- **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 [42], 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

[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



Turn on Living Room Lights
Turn on TV Set-top Box
Adjust NEST temperature
Order a Cappuccino
Get the Current Temperature
Turn off the Smart Plug for Coffee Machine

Save Operation Confirmation
Are you sure you want to record the operation: Click on the ActionBar$Tab object "LIGHTS" in Hue

EPIDOSITE Recording Panel
Click
Target Type
android.widget.FrameLayout
Identifying Features
☐ ViewID: com.dlink.mydlinkmyhome:id/imgIconStatus
Set as a parameter
✓ Text: Lab Motion Sensor
Set as a parameter
☐ ViewID: com.dlink.mydlinkmyhome:id/txt_title
Set as a parameter
☐ Location in Parent: 0 0 1080 253
☐ Location in Screen: 0 216 1080 469
Within App
mydlink Home
90 total alternative nodes, 1 matched
Preview
Click on the FrameLayout object "Lab Motion Sensor" in mydlink Home

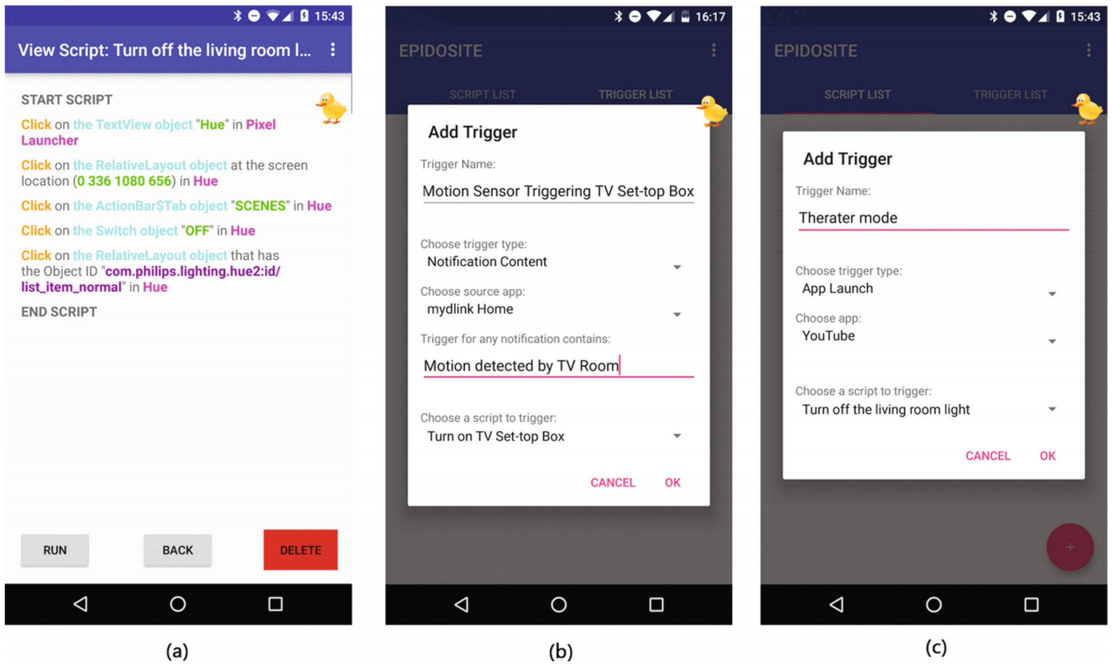
Figure 6.1: Screenshots of EPIDOSITE: (a) the main screen of EPIDOSITE showing a list of available scripts; (b) the confirmation dialog for an operation; (c) the data description editing panel for one operation. *Note that in the latest version, the data description editing panel (c) has been replaced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

use those familiar app interfaces to develop automations 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 [33, 93]. 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, 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 exactly 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



View Script: Turn off the living room l...

START SCRIPT

Click on the TextView object "Hue" in Pixel Launcher

Click on the RelativeLayout object at the screen location (0 336 1080 656) in Hue

Click on the ActionBar$Tab object "SCENES" in Hue

Click on the Switch object "OFF" in Hue

Click on the RelativeLayout object that has the Object ID "com.philips.lighting.hue2:id/list_item_normal" in Hue

END SCRIPT

Add Trigger

Trigger Name: Motion Sensor Triggering TV Set-top Box

Choose trigger type: Notification Content

Choose source app: mydlink Home

Trigger for any notification contains: Motion detected by TV Room

Choose a script to trigger: Turn on TV Set-top Box

Add Trigger

Trigger Name: Therater mode

Choose trigger type: App Launch

Choose app: YouTube

Choose a script to trigger: Turn off the living room light

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.

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.

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 [37, 148]. 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 intercepts 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), 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 [22, 68] and also varies [159] 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 services 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

[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.

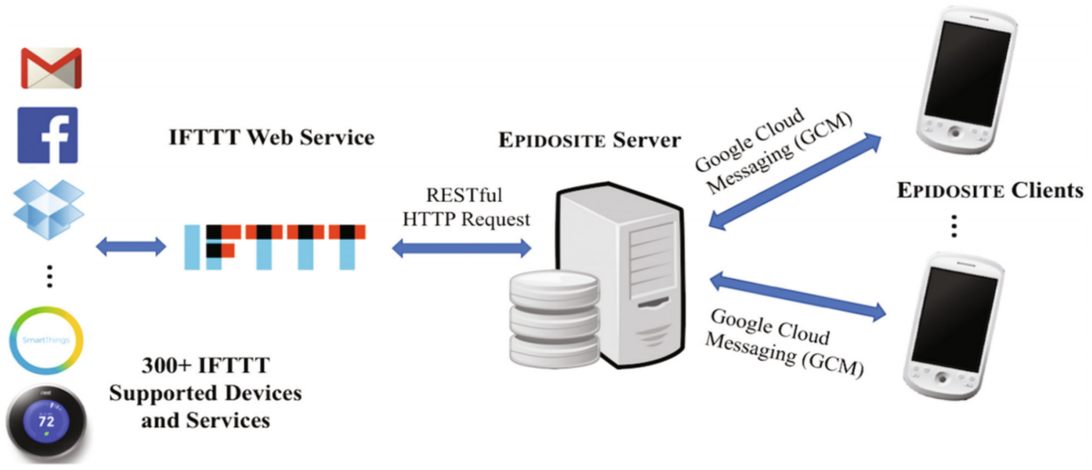
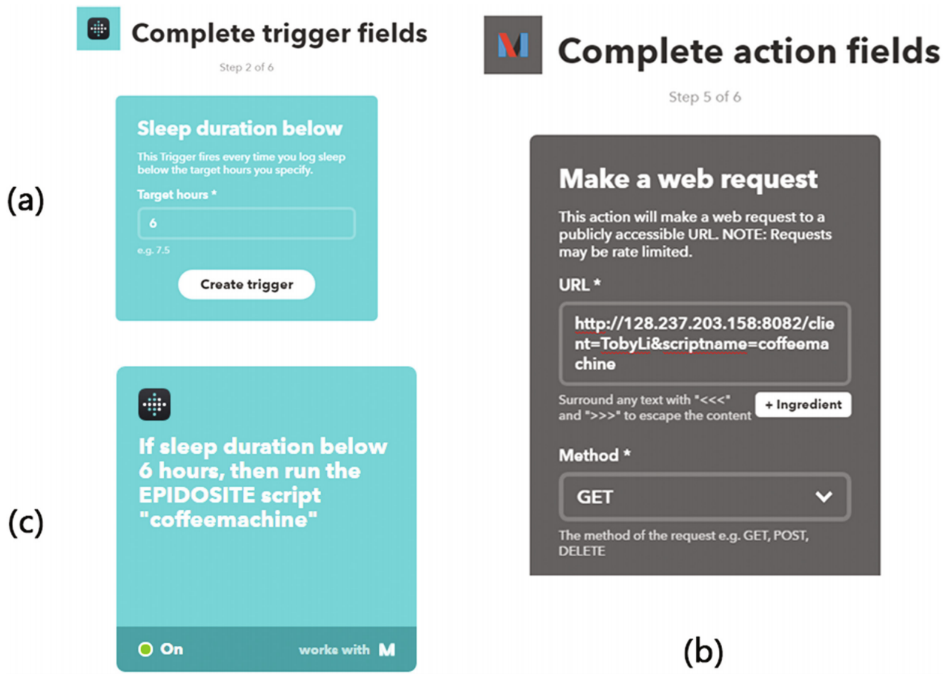

IFTTT Web Service
EPIDOSITE Server
RESTful HTTP Request
Google Cloud Messaging (GCM)
EPIDOSITE Clients
Google Cloud Messaging (GCM)
300+ IFTTT Supported Devices and Services
IFTTT

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



Sleep duration below

This Trigger fires every time you log sleep below the target hours you specify.

Target hours *

6

e.g. 7.5

Make a web request

This action will make a web request to a publicly accessible URL. NOTE: Requests may be rate limited.

URL *

http://128.237.203.158:8082/client=TobyLi&scriptname=coffeemachine

Surround any text with "<<<" and ">>>" to escape the content

Method *

GET

The method of the request e.g. GET, POST, DELETE

If sleep duration below 6 hours, then run the EPIDOSITE script "coffeemachine"

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.

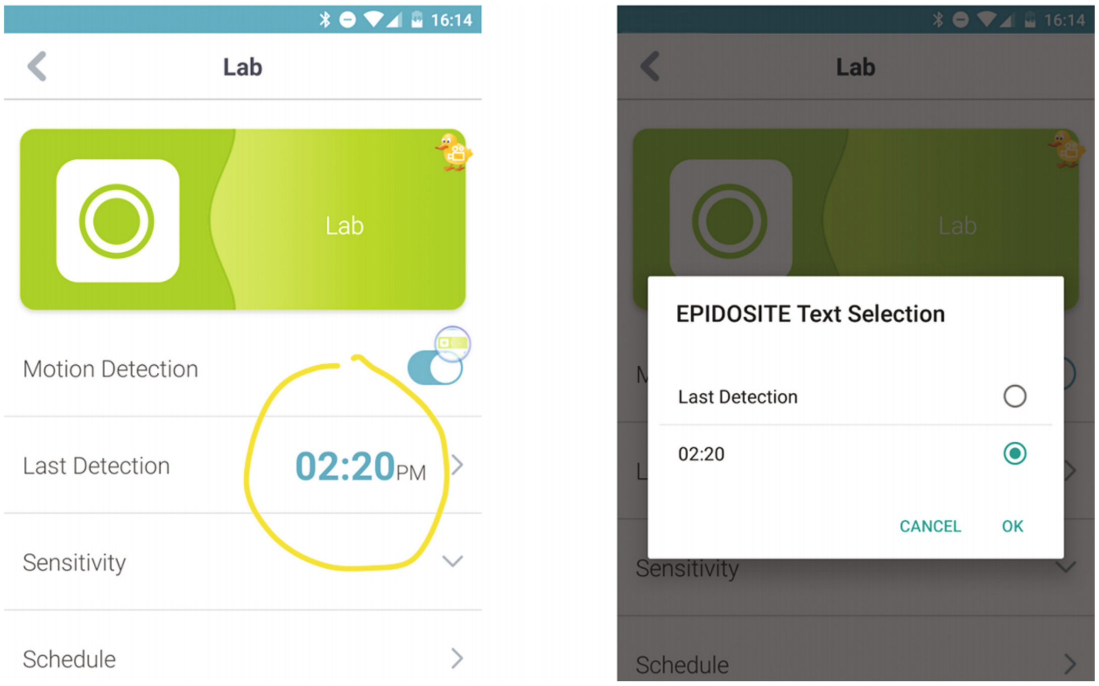


Figure 6.5: Extracting the time of the last detection from a D-link motion sensor in the Mydlink Home app using a circle gesture in EPIDOSITE.

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 an long-time important challenge [42]. Sharing data across devices from different "ecosystems" often requires the user to manually setup the connection using techniques like HTTP requests, which require carefully planning and extensive technical expertise [36]. Middleware like [16,49,71,141] 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 the an app by using a gesture during the demonstration, storing the value in a variables 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.

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 any 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 an 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 automations need

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

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

to run in the foreground on the main UI thread of the phone. Thus, if an 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, the separate phone can be consistently plugged in and stay connected with the IoT devices to ensure that the automations 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 automations 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

Proposed Work

In our work thus far, we have designed and developed various mechanisms, interfaces, and techniques to better support end users to extend, customize, and appropriate intelligent agents for their own app-based computing tasks. In particular, the systems have made contributions on the usability, applicability, generalizability, and flexibility of end user development for task automation. These components have been consolidated in an integrated prototype of the SUGILITE system, with which we have conducted several lab usability studies.

The shared theme of my proposed work is to make SUGILITE more robust, practical, and powerful. At the end, I also plan to deploy it with a group of actual users in the field. Specifically, I plan to *(i)* improve SUGILITE's robustness by boosting its support for understanding verbal instructions with diverse and flexible structures, expressions, and styles, *(ii)* design new interfaces for supporting effective error handling in the user's interaction with SUGILITE, *(iii)* address privacy concerns in sharing SUGILITE scripts by developing a new approach for identifying and obfuscating private personal information embedded in end-user-developed PBD scripts, and *(iv)* deploy SUGILITE with a group of actual users in the field, from which we can study the usage of SUGILITE in real contexts, as well as evaluate its *usefulness* in real-life scenarios.

7.1 Understanding Diverse and Flexible Verbal Instructions

In its current prototype, SUGILITE has limited accuracy for understanding diverse and flexible verbal instructions. In the previous lab usability studies, it sometimes misunderstood user utterances with different vocabulary, expressions, or syntax. This issue limits SUGILITE's robustness, and can be especially problematic for the planned field deployment study.

To address this problem, I plan to work on boosting SUGILITE's capability of understanding diverse and flexible verbal instructions. As discussed in Sections 4.3.2, and 5.4.1, the current SUGILITE prototype uses a grammar-based floating parser trained with a small corpus with less than a thousand examples. Therefore, it sometimes cannot correctly understand alternative ways of expressing the same intents due to the limited grammar rules and training data. For example, it can understand "when the current weather is rainy" and "if the price is higher than $10", but not "when it rains" or "if I need to pay more than $10" because the current grammar needs to extract concrete target concepts like "current weather" and "price" to operate correctly.

The first step towards addressing this problem is to collect a large set of sample utterances that can represent diverse and flexible vocabulary, expressions, or syntax that users might use when interacting with different parts SUGILITE. Using this dataset, we can determine the technical approach to use — whether it is sufficient to construct new grammar rules to cover most expressions used by SUGILITE users, or if SUGILITE needs a different parsing architecture. After making the improvements, we will train the new semantic parser and evaluate its performance using the collected dataset.

One thing I want to clarify is that improving semantic parsing performance is a long-standing and difficult NLP problem, and **not** a research focus of this dissertation. The expected outcome of this part of the work is to produce a more robust language understanding mechanism that can handle diverse and flexible user verbal instructions that SUGILITE may encounter in the planned field deployment study.

7.2 Better Support for Error Handling

To make SUGILITE robust, another crucial step is to enable it to properly identify, handle, and recover from errors. There have been some error handling mechanisms developed in SUGILITE: Chapter 3 discusses how SUGILITE identifies runtime errors caused by app updates or unexpected app states, and allows the user to fix the script by either creating a fork to handle the new situation, or replacing a part of the script when the underlying app has permanently changed. Chapter 5 describes how SUGILITE allows the user to fix ambiguity errors when it cannot understand specific comparators (e.g., in a condition "when the score is better than 3", whether "better" means "greater than" or "smaller than"), and how SUGILITE's conversational framework allows the user to backtrack to a previous conversational state. The PBD approach used in SUGILITE also addresses the "out-of-domain" error commonly seen in prevailing conversational intelligent agents by allowing end users to "teach" procedures and concepts that are unknown to agents [87].

Despite progress, many types of errors across SUGILITE's pipeline remain under-addressed. In order to make SUGILITE robust enough for the field deployment, I propose to explore the design space of error handling for conversational end-user-programmable intelligent agents, and design new error handling techniques in SUGILITE.

7.2.1 The Sources of Potential Errors in SUGILITE

Besides those that have been covered in previous chapters, I identify the following sources of potential errors to address for the proposed work:

1. **Speech recognition:** When a user gives a speech command, the agent needs to transcribe the user's speech input into text. During the speech-to-text process, the agent may either not hear or mishear what the user has said.

2. **Intent classification:** After speech recognition, the agent needs to classify the user's input into a task intent. Sometimes the user input is inherently ambiguous (e.g., "book me a room" can mean either booking a hotel room, or reserving a conference room). At other times, the agent may also have problem handling the user's diverse expressions not covered in the original utterance (e.g., when the user says "find me a place to stay" instead of "book a hotel room"). Another specific challenge for end-user-programmable agents like SUGILITE is that since the user can program new tasks, the agent needs to also correctly identify when the user has talked about a previously unknown new intent.

3. **Entity resolution:** From the user input, the agent also needs to identify entities mentioned in the utterance, and resolve them to known entities in its knowledge base. Some entities can be ambiguous.

For example, when the user says "What's the price of *apple*?", the word "apple" can be resolved to either the stock of Apple Inc., or the apple fruit.

The last two types of errors are more predominant in end-user-programmable agents like SUGILITE compared with prevailing conversational intelligent agents. Accurate intent classification and entity resolution often rely on large training corpora. However, since task intents and relevant entities in SUGILITE are learned directly from end users, the system usually has only one, or very few example utterances for each specific task domain. Therefore SUGILITE is more likely to be confused about intents and entities than intelligent agents with professionally developed "skills".

7.2.2 Proposed Approach

When we ask a user to verbally explain an ambiguous intent or an entity in natural language, the user may very likely refer to more intents, entities, reasonings and concepts that the system does not know, or is not sure about. This is especially true in EUD scenarios where the agent has little existing knowledge about the domain. However, when an user refers to a specific app on the phone, shows a screen in the app, or highlights a piece of content within the app, I hypothesize that the agent can leverage this information, together with all the rich contextual data within the app GUI, to disambiguate the original input.

The previous work provides preliminary evidence in support of this hypothesis — Chapter 4 shows that GUIs of mobile apps are an effective medium for users to clarify their intentions for specific demonstrated actions (e.g., clicking on a button) on GUIs. In the proposed work, I seek to apply a similar technique for disambiguation and error handling on the task level. We will design and develop a new multi-modal mixed-initiative interface to allow users to clarify their inputs by referring to relevant apps, app screens, or app contents to repair errors in speech recognition, intent classification, or entity resolution when interacting with end-user-programmable intelligent agents like SUGILITE.

7.2.3 Proposed Evaluation Methods for Error Handling

While I am still at an early stage of exploring the design space for this new interface, I expect to evaluate the outcome of the proposed work using the following metrics:

1. Can the proposed interface enable users to recall, find, and provide appropriate references to apps and their contents to help handle speech recognition, intent classification, and entity resolution errors encountered in their interactions with SUGILITE?

2. With the app references provided by users, can the proposed technique help SUGILITE successfully recover from the aforementioned types of errors?

To answer these two questions, I plan to run a task-based lab study. First, we will give users some sample error scenarios, and test how well they can recall, find, and provide appropriate app references relevant to the task with the help of the proposed interface. Second, we will assess whether the proposed system can successfully help SUGILITE recover from the sample error scenarios using the dataset of provided inputs from the user study.

7.2.4 Related Work

In conversational systems, user utterances are frequently misheard, misunderstood, or missed. Users often borrow error-repairing strategies from human-human interactions when interacting with systems, both on the verbal level (e.g., repetition, semantic adjustments, and syntactical adjustments) and the paraverbal level (e.g., prosodic changes, over-articulation and increased volume) [11, 20, 130]. However, these strategies are sometimes not as effective in human-agent interactions due to technical limitations [132]. In practice, users often just adapt their communication patterns based on their perceived capabilities of the system, as they gain more experience with it [69].

Various UX design guidelines have been proposed (e.g., [4,51]) to help developers of conversational systems to better handle errors. Some example strategies include designing task specific fallback/clarification questions, providing possible options, and providing contextual help [4]. However, those methods rely on knowledge about the context of the underlying task, which require developer effort for each specific task domain, making it not applicable in SUGILITE's EUD use case where users may perform tasks in any new domains. ScratchThat [153] supported command-agnostic speech repair for parameters in spoken commands. It focused on corrective clauses that fixed problems in original utterances (e.g., dealing with situations when the user said the wrong word) while my proposed work will focus on errors made by the agent.

Our proposed approach uses a multi-modal approach. The idea of leveraging additional input modalities has been explored in prior work. For example, spelling a word can be an effective way to correct speech recognition errors [110]. Pen or touch-based inputs can also be useful for locating and repairing errors in speech systems [127, 146]. But these systems can only deal with errors on a low-level (e.g., dictation) but not those regard to task semantics. Compared with the previous work, our proposed approach leverages mobile apps and their GUIs as the medium so that it *(i)* can work with diverse and long-tail task domains without requiring development effort for each supported domain, and *(ii)* can deal with errors regarding specific task intents and entities.

7.3 Privacy Preservation in Script Sharing

SUGILITE focuses on supporting end user development (EUD), which according to a popular definition, refers to programming activities to achieve a result primarily intended for personal, rather public use [74]. However, despite the primary intention, sharing of the program artifacts is common in EUD (e.g., [101]). The CoScripter [82] paper outlined two main motivations for sharing end user developed task automation scripts: *(i)* for tasks that are performed frequently across multiple users and are tedious, the main motivation is efficiency; and *(ii)* for other tasks that are complex or hard to remember, the shared scripts are also used as the medium for sharing "how-to" knowledge between users. A later field study of CoScripter scripts revealed that users have shared CoScripter scripts for automating a wide range of both work-related and non-work-related tasks [17], which affirmed the need for sharing end-user developed scripts.

While the support for script sharing in EUD is useful and desired, its adoption has been limited. A major barrier to the wide-adoption is the privacy problem [17, 82, 84]. To address this problem, I propose to design and implement a new mechanism for supporting privacy-preserving script sharing in SUGILITE as a part of my dissertation.

7.3.1 The Privacy Problem in Sharing EUD Scripts

When sharing EUD artifacts, users are often concerned about privacy. More specifically, they do not want to leak any personal information that may get accidentally embedded in the shared artifacts. This issue is particularly important in GUI-based programming by demonstration (PBD) systems such as SUGILITE, since in such systems end-user developers do **not** directly author the programs, but instead demonstrate one or more examples of the desired system behaviors on the graphical user interfaces (GUIs) of target applications. As a result, end-user developers have less knowledge and control about what has been included in the resulting program, and therefore are hesitant about sharing them [17, 82, 84]. The privacy concerns in EUD script sharing have been identified in several studies. Email and interview studies for CoScripter [82] found that the privacy concern was a main barrier to sharing. Participants reported that they decided to make scripts private because they worried about having private personal information embedded in scripts. Studies and discussions for other EUD systems such as ActionShot [84] and RePlay [47] have also identified similar privacy concerns.

In SUGILITE, the system not only collects information about the exact demonstrated actions, but also rich contextual information (e.g., sensor readings and the contents of the screens including the app's responses), and sometimes conversational data from the user, in order to better infer the users' intents from the demonstrations, to better support error detection by making sure that the user is on the correct page, and to further generalize the resulting programs. While these new mechanisms make the PBD more powerful, flexible and robust, they also increase the likelihood of accidentally including personal information in program artifacts, and made it infeasible for end-user developers to manually scrutinize the program artifacts before sharing them, since the collected contextual information is not currently shown anywhere because of its overwhelmingly large amount.

Additional constraints related to identifying and removing personal information from program artifacts created from GUI demonstrations is to preserve the scripts' transparency and readability. In task automation, it is important to enable people who want to use a script shared by another to be able to fully understand its behavior, so that they can *(i)* select the correct script for their needs, *(ii)* identify any potentially dangerous or malicious operations in the script, and *(iii)* modify or extend the script when needed. As a result, my proposed approach aims to avoid unnecessary obfuscation of the shared scripts that hinders transparency and readability. The proposed approach should also preserve as much meta information from demonstrations as possible in shared scripts, since such meta information is very useful for the robustness, extensibility and generalizability of the shared scripts.

7.3.2 Related Work

Prior approaches for identifying personal private information in shared EUD scripts have limitations. CoScripter [82] used a "personal database" where end user developers 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 user's personal database. An apparent downside of this approach is that it requires users to manually create such personal databases, which can require significant effort. Personal databases also only cover the most common and obvious types of personal information, and only work when database entries show up as exact string matches in shared scripts. Another approach is to represent operations in scripts as lists of textual steps (e.g., ActionShot [84]) or in visual programming blocks (e.g., Rousillon [26] and Helena [24]) before sharing, and ask users to verify if any private personal information is included. This approach also requires manual intervention from users, and is not scalable to SUGILITE, which collects rich contextual information along with the operations.

My proposed work is also related to prior literature from the privacy and security communities on detecting personal information leaks in shared data. For example, systems like PrivacyProxy [142], Agrigento [30], and LeakDoctor [152] can detect leaks of personal information by analyzing the contents of outbound network packets. Agrigento [30] and LeakDoctor [152] 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 types of personal information. My proposed approach for identifying personal information leaks is similar to that of PrivacyProxy [142] – both systems identify private personal information based on the uniqueness of the data. This approach does not limit the types of personal information that it can detect, providing coverage even for uncommon types of information.

Another way to protect personal data is to monitor when, how, and what granularity of personal data are accessed and shared by apps [91]. 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 apps. In contrast, since my proposed approach 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).

7.3.3 Design Goals

Based on the prior work in this area and the high-level design goals of SUGILITE, I identify the following design goals for the proposed new mechanism:

1. It should require minimal user effort and intervention; in particular, it should not require users to manually tag personal information from the large amount of collected data.

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

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

4. It should work with PBD scripts on non-specific task domains and third-party app GUIs.

5. To ensure data security, sensitive personal information should never 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).

7.3.4 Proposed Approach

I plan to explore the approach of identifying private personal information in shared scripts and their corresponding contextual and meta data based on the *uniqueness* of information. Specifically, I consider an *information entry* as a pair of the content of a piece of information and the context where the information is displayed. The intuition is that information entries only appear on one, or a very small number of users' devices are likely personal, while others that appear across devices of many users are likely not.

This approach responds to the emerging challenge of preventing *re-identification attacks* [43] when protecting identifiable personal information. In some 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 weather at the user's location, the number of Starbucks stores within the service area, and the estimated store wait time from a user-demonstrated script of ordering coffee using the Starbucks app, one can likely infer private information, such as the user's location, fairly accurately. Therefore, masking only explicit personal information is not sufficient. My proposed technique identifies and masks *all* information entries that are unique or uncommon among all users, making re-identification attacks much more difficult and infeasible.

To operationalize this concept, We will use the context of the mobile app to approximate the context of the information. That is, an information entry indicates "the information X is displayed on the screen Y in the app Z." This representation captures the role of context in determining whether a piece of information is private. For example, the string "New York" showing up on the checkout screen of the Starbucks app can be private personal 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.

For collecting data, we will set up a server that collects and aggregates hashed information entries in the background from app GUIs of many users. On the client side, when not recording, the system collects app context-content pairs in the background, and uploads their hash values to the server. The server side keeps count of the number of unique users with identical information entries (i.e., how many users have had 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). The counts are calculated using the one-way hash of information entries and app context, thus the server cannot determine their original values.

Before uploading a script for sharing, the script developer's client side looks up the aggregated results on the server for each information entry included in the script to determine its uniqueness. When an information entry is labeled unique, the system will upload it in its hashed form. Obfuscating string values in data description queries or parameter values by hashing does **not** prevent executing scripts in their current forms — data description queries in SUGILITE rely on comparing the *equality* of string values (e.g., click on the button that has text "next"), which is supported with hash values. The comparator `contains` can also be supported on the token level (e.g., click on the button that has text that contains the token "profile") by breaking down strings into tokens when generating UI Snapshot Graphs in SUGILITE.

Despite the fact that scripts with obfuscated information can still be executed, I plan to develop the mechanism for recovering the obfuscated information in scripts *locally* as much as possible on the script user's client side because *(i)* revealing information entries used in script data descriptions helps improve script transparency, allowing users to understand script behaviors, to validate if they fit the user need, and to modify and extend the scripts when needed, and *(ii)* when the script parameters values are personal to the original developer, reconstructing them on the script user's end allowing the user to take advantage of the parameterization with their own personal options.

7.3.5 Proposed Evaluation Methods for Sharing and Privacy

I plan to evaluate this proposed work using the following three metrics:

1. Does the proposed mechanism accurately identify and obfuscate private personal information in end-user-developed SUGILITE scripts?
2. How do users perceive the level of privacy provided by the proposed mechanism? Can this mechanism encourage them to be more comfortable with sharing SUGILITE scripts they have developed?
3. Does the proposed mechanism leave sufficient information in the resulting SUGILITE scripts to retain their transparency and readability?

For metric 1, we will conduct an offline evaluation, where we simulate recording, executing, and sharing scripts for a diverse set of popular tasks over many mobile apps. Then we can compare the private personal information identified by the proposed mechanism against the ground-truth result labeled by experts to assess the accuracy, precision, and recall of our approach in identifying and obfuscating private personal information.

For metrics 2 and 3, we will conduct a user study. I plan to develop a survey to test the user's perceived level of privacy, and a set of sample tasks to assess the transparency and readability of PBD scripts. From the survey and the sample tasks, we evaluate how the proposed mechanism compares with three baseline conditions: *(i)* obfuscating no information, *(ii)* obfuscating only common personal information (e.g., phone number, address), and *(iii)* obfuscating *all* strings.

7.4 Field Deployment Study

To complete this dissertation, I will deploy SUGILITE to a group of actual users as a way to *(i)* validate the feasibility and robustness of the system with the new features I am proposing, *(ii)* measure the usefulness of SUGILITE in real-life scenarios, and *(iii)* study the characteristics of how users use SUGILITE. The key goal of the deployment is to study SUGILITE within its intended context of use [140]. Through the deployment, we will also collect a useful dataset of SUGILITE scripts created by end users, together with the demonstrations and natural language instructions for creating them, their usage data, and the patterns of how they are shared among users (if any).

7.4.1 Research Questions

I plan to answer the following research questions through the proposed study:

RQ1. Is SUGILITE sufficiently robust to handle diverse tasks and contexts in the field?

RQ2. Do users find SUGILITE useful for their own task automation needs?

RQ3. What kinds of scripts do users create with SUGILITE?

RQ4. How do users create these scripts? For example, how do users naturally coordinate the two modalities? What SUGILITE features do they use?

RQ5. What kinds of contexts do users use SUGILITE in?

RQ6. How and when do users generalize, extend, modify and share their SUGILITE scripts?

7.4.2 Proposed Approach

In order to answer the above research questions, I plan to deploy SUGILITE to a group of around 20–30 users for an extended period of time (at least a month). This group should all be proficient and active Android smartphone users who *(i)* are familiar with mobile apps, and *(ii)* frequently use mobile apps for various tasks. The group should also represent users with different levels of programming expertise and experience with intelligent agents, and users with different app usage behaviors [159] (e.g., those who use apps mostly for personal tasks vs. those who use apps for professional tasks).

We will use a mix of qualitative (e.g., interviews, surveys, experience sampling (ESM)) and quantitative (e.g., log data analysis) methods to investigate the proposed research questions. Enabled by the privacy-preserving script sharing mechanism proposed in Section 7.3, we will also collect anonymized versions of SUGILITE scripts created by users, their usage and sharing statistics, and the corresponding meta-data indicating the contexts of their creation and invocation.

7.5 Timeline for Completion

My goal is to complete the dissertation by May 2021. Table 7.1 shows a timeline of my plan. I will work with undergraduate and master research assistants who will help implement the planned features, perform the user studies, and analyze the results.

Timeline	Work
Oct. 2019–Mar. 2020	Developing and evaluating the privacy-preserving script sharing mechanism
Nov. 2019–Apr. 2020	Developing and evaluating the new interface for error handling
Nov. 2019–Apr. 2020	Developing and evaluating the new technique for understanding verbal instructions
May 2019–Aug. 2020	Running the field deployment study
Aug. 2020–Sep. 2020	Analyzing study data
Oct. 2020–Apr. 2021	Writing the dissertation
May 2021	Defending the dissertation

Table 7.1: Plan for completion of my dissertation

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] Amazon. 2019. Amazon Alexa Design Guide. (2019). https://developer.amazon.com/docs/alexa-design/get-started.html

[5] V. Antila, J. Polet, A. Lms, 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

[6] 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

[7] 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.

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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.

[13] 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

[14] 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.

[15] 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`

[16] 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/`

[17] 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`

[18] 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`

[19] 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.

[20] Holly P Branigan, Martin J Pickering, Jamie Pearson, and Janet F McLean. 2010. Linguistic alignment between people and computers. *Journal of Pragmatics* 42, 9 (2010), 2355–2368.

[21] 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`

[22] Matthias Bhmer, Brent Hecht, Johannes Schning, Antonio Krger, 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`

[23] 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*.

[24] 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.

[25] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018a. 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`

[26] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018b. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology*. ACM, 963–975.

[27] 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`

[28] 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.

[29] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[30] 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*.

[31] 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`

[32] 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`

[33] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.

[34] 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`

[35] 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

[36] 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 Daz, 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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.

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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.

[49] 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

[50] Cristian Gonzlez Garca, B. Cristina Pelayo G-Bustelo, Jordn 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

[51] Google. 2019. Conversation design | Actions on Google. (2019). https://developers.google.com/actions/design/

[52] 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`

[53] 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`

[54] H Paul Grice, Peter Cole, Jerry Morgan, and others. 1975. Logic and conversation. *1975* (1975), 41–58.

[55] Dominique Guinard and Vlad Trifa. 2009. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, Vol. 15. `http://webofthings.org/2009/04/20/web-mashups-mem/`

[56] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. 2009a. 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/`

[57] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. 2009b. 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/`

[58] 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`

[59] 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.

[60] Daniel C. Halbert. 1993. SmallStar: programming by demonstration in the desktop metaphor. In *Watch what I do*. MIT Press, 103–123.

[61] 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`

[62] 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`

[63] 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.

[64] 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`

[65] IFTTT. 2016. IFTTT: connects the apps you love. (2016). `https://ifttt.com/`

[66] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. 2017. Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces (IUI '17)*. ACM, New York, NY, USA, 233–243. `DOI:http://dx.doi.org/10.1145/3025171.3025176`

[67] 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`

[68] 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`

[69] 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 (SIGIR '13)*. ACM, New York, NY, USA, 143–152. `DOI:http://dx.doi.org/10.1145/2484028.2484092`

[70] 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.

[71] 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`

[72] 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`

[73] James Kirk, Aaron Mininger, and John Laird. 2016. Learning task goals interactively with visual demonstrations. *Biologically Inspired Cognitive Architectures* 18 (2016), 1–8.

[74] 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`

[75] 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.

[76] 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.

[77] 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`

[78] 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`

[79] 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`

[80] 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`

[81] 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`

[82] 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`

[83] Hao Li, Yu-Ping Wang, Jie Yin, and Gang Tan. 2019b. SmartShell: Automated Shell Scripts Synthesis from Natural Language. *International Journal of Software Engineering and Knowledge Engineering* 29, 02 (2019), 197–220.

[84] 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`

[85] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017a. 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, 6038–6049. `DOI:http://dx.doi.org/10.1145/3025453.3025483`

[86] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Tom M. Mitchell, and Brad A. Myers. 2018a. 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)*.

[87] Toby Jia-Jun Li, Igor Labutov, Brad A. Myers, Amos Azaria, Alexander I. Rudnicky, and Tom M. Mitchell. 2018b. Teaching Agents When They Fail: End User Development in Goal-oriented Conversational Agents. In *Studies in Conversational UX Design*. Springer.

[88] 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.

[89] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019a. 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

[90] 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.

[91] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2017b. 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

[92] Percy Liang, Michael I. Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics* 39, 2 (2013), 389–446.

[93] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

[94] Henry Lieberman and Hugo Liu. 2006. Feasibility studies for programming in natural language. In *End User Development*. Springer, 459–473.

[95] 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

[96] 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

[97] 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

[98] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 2018. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. *CoRR* abs/1802.08802 (2018). http://arxiv.org/abs/1802.08802

[99] LlamaLab. 2016. Automate: everyday automation for Android. (2016). http://llamalab.com/automate/

[100] 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

[101] 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

[102] 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).

[103] 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

[104] 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.

[105] 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

[106] 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

[107] Rodrigo de A. Maus 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

[108] 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

[109] 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

[110] Arthur E McNair and Alex Waibel. 1994. Improving recognizer acceptance through robust, natural speech repair. In *Third International Conference on Spoken Language Processing*.

[111] 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

[112] 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

[113] 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.

[114] 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.

[115] 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

[116] 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.

[117] 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

[118] Brad A. Myers. 1993. Peridot: creating user interfaces by demonstration. In *Watch what I do*. MIT Press, 125–153.

[119] 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.

[120] 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

[121] 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

[122] 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

[123] 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

[124] Sharon Oviatt. 1999a. Mutual disambiguation of recognition errors in a multimodel architecture. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 576–583.

[125] 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

[126] Sharon Oviatt and Philip Cohen. 2000. Perceptual user interfaces: multimodal interfaces that process what comes naturally. *Commun. ACM* 43, 3 (2000), 45–53.

[127] Sharon Oviatt, Phil Cohen, Lizhong Wu, Lisbeth Duncan, Bernhard Suhm, Josh Bers, Thomas Holzman, Terry Winograd, James Landay, Jim Larson, and others. 2000. Designing the user interface for multimodal speech and pen-based gesture applications: state-of-the-art systems and future research directions. *Human-computer interaction* 15, 4 (2000), 263–322.

[128] 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

[129] 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.

[130] Jamie Pearson, Jiang Hu, Holly P. Branigan, Martin J. Pickering, and Clifford I. Nass. 2006. Adaptive Language Behavior in HCI: How Expectations and Beliefs About a System Affect Users' Word Choice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. ACM, New York, NY, USA, 1177–1180. DOI:http://dx.doi.org/10.1145/1124772.1124948

[131] 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

[132] 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 640, 12 pages. DOI:http://dx.doi.org/10.1145/3173574.3174214

[133] 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

[134] 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

[135] 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

[136] Andr 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

[137] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjrg 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

[138] Albrecht Schmidt. 2015. Programming Ubiquitous Computing Environments. In *End-User Development (Lecture Notes in Computer Science)*, Paloma Daz, 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

[139] 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.

[140] 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

[141] 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

[142] 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

[143] 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.

[144] Anselm Strauss and Juliet M. Corbin. 1990. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc.

[145] Bernhard Suhm, Brad Myers, and Alex Waibel. 2001a. 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

[146] Bernhard Suhm, Brad Myers, and Alex Waibel. 2001b. 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

[147] 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

[148] 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

[149] 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

[150] David Vadas and James R Curran. 2005. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*. 191–199.

[151] 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.

[152] 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

[153] Jason Wu, Karan Ahuja, Richard Li, Victor Chen, and Jeffrey Bigham. 2019. ScratchThat: Supporting Command-Agnostic Speech Repair in Voice-Driven Assistants. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 2 (2019), 63.

[154] 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

[155] 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

[156] 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.

[157] 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

[158] 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

[159] 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

[160] 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