APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions

Toby Jia-Jun Li[1], Igor Labutov[2], Xiaohan Nancy Li[3], Xiaoyi Zhang[5], Wenze Shi[3], Wanling Ding[4], Tom M. Mitchell[2], Brad A. Myers[1]

[1]HCI Institute, [2]Machine Learning Dept., [3]Computer Science Dept., [4]Information Systems Dept.
Carnegie Mellon University, Pittsburgh, PA, USA
{tobyli, ilabutov, tom.mitchell, bam}@cs.cmu.edu
nancylxh14@gmail.com, {wenzes, wanlingd}@andrew.cmu.edu

[5]Computer Science &Engineering
University of Washington
Seattle, WA, USA
xiaoyiz@cs.washington.edu

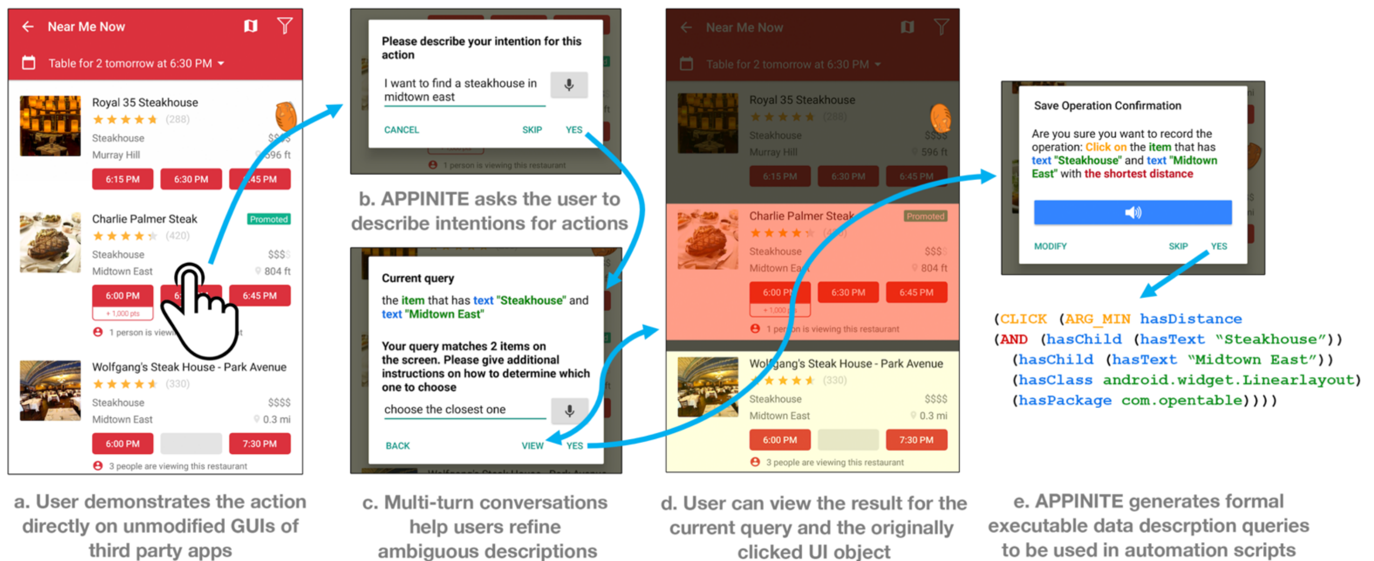


Fig. 1. Specifying data description in programming by demonstration using APPINITE: (a, b) enables users to naturally express their intentions for demonstrated actions verbally; (c) guides users to formulate data descriptions to uniquely identify target GUI objects; (d) shows users real-time updated results of current queries on an interaction overlay; and (e) formulates executable queries from natural language instructions.

***Abstract*— A key challenge for generalizing programming-by-demonstration (PBD) scripts is the *data description problem* – when a user demonstrates performing an action, the system needs to determine features for describing this action and the target object in a way that can reflect the user's intention for the action. However, prior approaches for creating data descriptions in PBD systems have problems with usability, applicability, feasibility, transparency and/or user control. Our APPINITE system introduces a multi-modal interface with which users can specify data descriptions verbally using natural language instructions. APPINITE guides users to describe their intentions for the demonstrated actions through mixed-initiative conversations. APPINITE constructs data descriptions for these actions from the natural language instructions. Our evaluation showed that APPINITE is easy-to-use and effective in creating scripts for tasks that would otherwise be difficult to create with prior PBD systems, due to ambiguous data descriptions in demonstrations on GUIs.**

Keywords—programming by demonstration, end user development, verbal instruction, multi-modal interaction, natural language programming

I. INTRODUCTION

Enabling end users to program new tasks for intelligent agents has become increasingly important due to the increasing ubiquity of such agents residing in "smart" devices such as phones, wearables, appliances and speakers. Although these agents have a set of built-in functionalities, and most provide expandability by allowing users to install third-party "skills", they still fall short in helping users with the "long-tail" of tasks and suffer from the lack of customizability. Furthermore, many of users' tasks involve coordinating the use of multiple apps, many of which do not even provide open APIs. Thus, it is unrealistic to expect every task to have a "skill" professionally made by service providers or third-party developers.

The lack of end-user programmability in intelligent agents results in an inferior user experience. When a user gives an out-of-domain command, the current conversational interface for most agents would either respond with a generic error message (e.g., "*sorry, I don't understand*") or perform a generic fallback action (e.g., a web search using the input as the search string). Often, neither response is helpful – a more natural and more useful response would be to ask the user to instruct the agent

This work was supported in part by Oath through the InMind project.

978-1-5386-4235-1/18/$31.00 ©2018 IEEE

how to perform the new task [1]. Such end-user programmability also enables users to automate their repetitive tasks, reducing their redundant efforts.

Programming by demonstration (PBD) has been moderately successful at empowering end user development (EUD) of simple task automation scripts. Prior systems such as SUGILITE [2], PLOW [3] and CoScripter [4] allowed users to program task automation scripts for agents by directly demonstrating tasks using GUIs of third-party mobile apps or web pages. This approach enables users to program naturally by using the same environments in which they already know how to perform the actions, unlike in other textual (e.g., [5], [6]) or visual programming environments (e.g., [7]–[9]) where users need to map the procedures to a different representation of actions.

The central challenge for PBD is generalization. A PBD system should produce more than literal record-and-replay macros (e.g., sequences of clicks and keystrokes), but learn the task at a higher level of abstraction so it can perform similar tasks in new contexts [10], [11]. A key issue in generalization is the *data description problem* [10], [12]: 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 Fig. 1a, 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 [13]. Some prior systems such as SmallStar [12] and Topaz [14] 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 SUGILITE [2], Peridot [15] and CoScipter [4] went a step further and used heuristic rules for generalization, which were still limited in applicability. This approach can only handle simple scenarios (unlike Fig. 1), and has the possibility of making incorrect assumptions.

At the other end of the spectrum, prior systems such as [16]–[20] 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 [21]. Even if end users provide a sufficient number of examples, prior studies [13], [22] 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 [21], resulting in a lack of trust, transparency and user control.

In this paper, we present a new multi-modal interface named APPINITE[1], based on our prior PBD system SUGILITE [2], to enable 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). We use an interaction proxy overlay in APPINITE to highlight ambiguous references on the screen, and to support meta actions for programming with interactive UI widgets in third-party apps.

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. Our 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.

II. BACKGROUND AND RELATED WORK

A. 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 [23]. APPINITE combines speech and touch to enable a "speak and point" interaction style, which has been studied since the early multi-modal systems like Put-that-there [24]. In programming, similar interaction styles have also been used for controlling robots (e.g., [25], [26]). A key pattern in APPINITE's multi-modal interaction model is *mutual disambiguation* [27]. When the user demonstrates an action on the GUI with a simultaneous verbal instruction, our system can reliably detect *what* the user did and *on which* UI object the user performed the action. The demonstration alone, however, does not explain *why* the user performed the action, and any inferences on the user's intent would be fundamentally unreliable. Similarly, from verbal instructions alone, the system may learn about the user's intent, but grounding it onto a specific action may be difficult due to the inherent ambiguity in natural language. Our system utilizes these complementary inputs to infer robust and generalizable scripts that can accurately represent user intentions in PBD.

A unique challenge for APPINITE is to support multi-modal PBD on arbitrary third-party GUIs. Some of such GUIs can be highly complicated with hundreds of objects, each with many different properties, semantic meanings and relationships with other objects. Moreover, third-party apps only expose low-level hierarchical representations of their GUIs at the presentation

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

layer, without information about internal program logic or semantics. Prior systems such as CommandSpace [28], Speechify [29] and PixelTone [30] investigated multi-modal interfaces that can map coordinated natural language instructions and GUI gestures to system commands and actions. But the use of these systems are limited to specific first-party apps and task domains, in contrast to APPINITE which aims to be general-purpose.

B. Generalization and Data Description Problems in PBD

Having accurate data descriptions to correctly reflect user intentions in different contexts is crucial for ensuring the generalizability in PBD. Prior PBD systems range from making no inference at all to using sophisticated AI algorithms to infer data descriptions for demonstrated actions [13].

The "no inference" approach (e.g., [12], [14]) shows dialogs to ask users to make selections on feature(s) to use for data descriptions when ambiguities arise, which gives users full control but suffers in usability because end users may have trouble understanding and choosing from the options, especially when the tasks are complicated, or when their intentions are non-trivial. The AI-based program synthesis approach (e.g., [16]–[20]) requires a large number of examples to cover the space of different contexts to synthesize from, which is not feasible in many cases when end users are unwilling to provide sufficient number of examples [21], or unable to provide high-quality examples with good coverage [13], [22]. Users also have limited control and understanding of the inference and synthesis process, as AI-based algorithms used in these systems often suffer in explainability and transparency [21].

APPINITE addresses these issues by providing a multi-modal interface to specify data descriptions verbally through a multi-initiative conversation. It provides users with control and transparency of the process, retains usability by allowing users to describe the data descriptions in natural language, provides increased expressive power in parsing natural language instructions, and eliminates redundancy by only requiring one example of demonstration and instruction.

C. Learning Tasks from Natural Language Instructions

Natural language instruction is a common medium for humans to teach each other new tasks. For an agent to learn from such instructions, a major challenge is grounding – the agent needs to extract semantic meanings from instructions, and associate them with actions, perceptions and logic [31]. This process is also related to the concept of natural language programming [32]. Some prior work has tried translating natural language directly to code (e.g., [33]–[35]), but these systems required users to instruct using inflexible structures and keywords that resemble those of the programming languages, which made such systems unsuccessful for end user developers.

In specific task domains such as navigation [36], email [31], robot control [37] or basic phone operations [38], the number of relevant actions and concepts are small, which makes it feasible to parse natural language into formal semantic representations in a smaller space of pre-defined actions and concepts.

An effective way to constrain user natural language instructions, but still support a wide variety of tasks, is to leverage GUIs of existing apps or webpages. PLOW [3] is a web automation agent that uses GUIs to ground natural language instruction. It asks users to provide "play-by-play" natural language instructions with task demonstrations, which is similar to APPINITE. PLOW grounds the instructions by resolving noun phrases to items on the screen through a heuristic search on the DOM tree of the webpage. SUGILITE [2], on the other hand, uses a single utterance describing the task from the user for each script to perform parameterization by grounding phrases in the initial utterance (e.g., order a cup of *cappuccino*) to a demonstrated action (e.g., select *cappuccino* from a list menu).

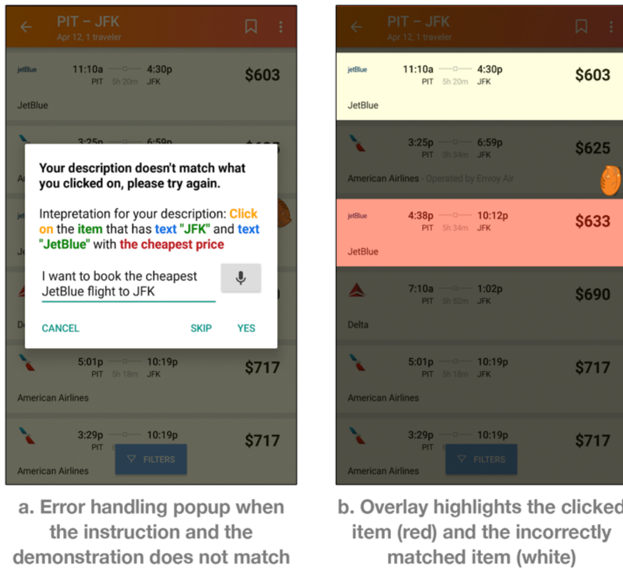
Compared with prior systems, APPINITE specifically focuses on helping users specify accurate data descriptions that reflect their intentions using a combination of natural language instructions and demonstrations. Our novel semantic relational graph representation of the GUI allows users to use a wider range of semantic (e.g. "*cheapest* restaurant") and relational (e.g., "score *for* Pittsburgh Steelers") expressions without being tied to the underlying GUI implementation. Users can also use more flexible logic in their instructions thanks to our versatile semantic parser. To ensure usability while giving the user full control, our mixed-initiative system can engage in multi-turn conversations with users to help them clarify and extend data descriptions when ambiguities arise.

III. 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 [39]) 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



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)

Fig. 2. APPINITE's error handling interfaces for handling situations where the instruction and the demonstration do not match.

abstracts the semantics and relationships from the platform-specific implementation of GUIs and maps more naturally to the semantics of likely natural language explanations.

IV. The APPINITE Interface

Informed by the results from the formative study, we have designed and implemented APPINITE to enable users to provide natural language instructions for specifying data descriptions in PBD. It uses our open-source SUGILITE [2] framework for detecting and replaying user demonstrations. APPINITE aims to improve the process for specifying data descriptions in PBD through its novel multi-modal interface, which provides end users with greater expressive power to create flexible programming logic while retaining a low learning barrier and high understandability. In this section, we discuss the user experience of APPINITE with an example walkthrough of specifying the data description for programming a script for making a restaurant reservation using the OpenTable app. Readers can also refer to the supplemental video figure for a similar example task.

A. *APPINITE User Experience*

After the user starts a new demonstration recording, she demonstrates clicking on the OpenTable icon on the home screen, and chooses the "Near Me Now" option on the main screen of OpenTable, which are exactly the same steps that she would do normally to make a restaurant reservation. Neither of these steps is ambiguous, because their data descriptions (clicking on the icon / FrameLayout object with text labels "OpenTable" / "Near Me Now") can be inferred using heuristic rules. Thus, the APPINITE disambiguation feature will not be invoked. Instead, the user directly confirms the recording either by speech or by tapping on a popup (Fig. 1e).

As the next action, the user chooses a restaurant from the result list (Fig. 1a). This action is ambiguous because its target UI object has multiple reasonable properties for data description, for which the heuristic-based approach cannot determine which one would reflect the user's intention. Therefore, APPINITE's interaction proxy overlay (details in Section V) prevents this tap from invoking the OpenTable app action, and asks the user, both vocally and visually through a popup dialog, to describe her intention for the action. The user can then either speak or type in natural language. Leveraging the UI snapshot graph extraction and natural language instruction parsing architecture (details in Section V), APPINITE can understand flexible data descriptions expressed in diverse natural language instructions. These descriptions would otherwise be impractical for end users to manually program. Below we list some example instructions that APPINITE can support for the GUI shown in Fig. 1a.

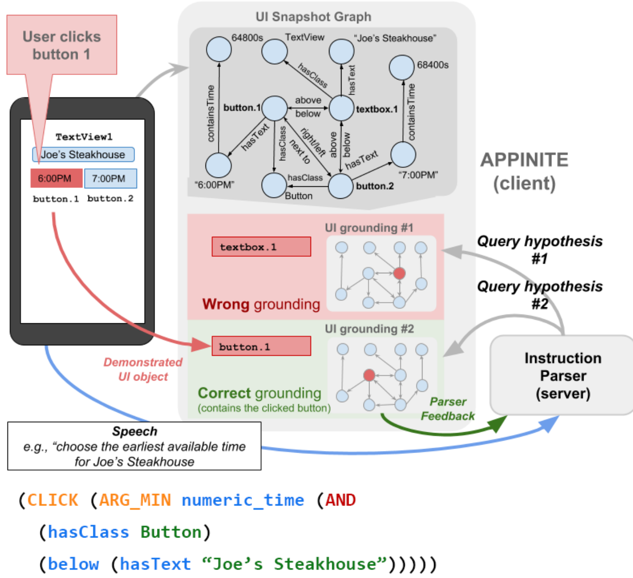
- *I want to choose the second search result*
- *Find the steakhouse with the earliest time available*
- *Here I'm selecting the closest promoted restaurant*
- *I will book a steakhouse in Midtown East*

End users might not be able to provide complete data descriptions to uniquely identify target UI objects on their first attempt. To address this issue, APPINITE uses a mixed-initiative multi-turn dialog interface to initiate follow-up conversations to help users refine data descriptions. For instance, as shown in Fig. 1c, the description parsed from the user's instruction matches two items in the list. APPINITE asks the user what additional criteria can be used to choose between the GUI objects when the initial query matches multiple ones. The user can preview the result of executing the current query on a screen captured from the underlying app's GUI (Fig. 1d). In this preview interface, the actually clicked object is marked in red, while the other matched ones (false positives) are highlighted in white. The user can iteratively refine the data description, add new requirements and preview the real-time result of the current data description until she has one that can both uniquely identify the action she has demonstrated and accurately reflects her intention.

Lastly, APPINITE formulates an executable data description query for the demonstrated action and adds it to the current automation script (Fig. 1e). This data description is used by the intelligent agent to choose the correct action to perform in future executions of the script in different contexts. The interaction proxy overlay then sends the previously held tap to the underlying app GUI, so that the app can proceed to the next step so the user can continue demonstrating the task.

In the above example, the user has interacted with the APPINITE interface in the "demonstration-first" mode where she first demonstrates the action, and only needs to provide natural language instructions to clarify her intention for the action if disambiguation is required. Alternatively, APPINITE's multi-modal interface also supports a "verbal-first" mode where she can first describe the action in natural language, after which she would only be asked to tap the correct UI object for grounding the data description if her description is ambiguous and matches multiple objects. All APPINITE interfaces used for recording are also speech-enabled, where users can freely choose the most natural interaction modality for the context – either direct manipulation, natural language instruction or a mix of both.

APPINITE also provides end-user-friendly error messages when the user's instruction does not match the demonstration




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

Fig. 3. APPINITE's instruction parsing process illustrated on an example UI snapshot graph constructed from a simplified GUI snippet.

(Fig. 2), or when the parser fails to parse the user's natural language instructions into valid data description queries. If a user has encountered the same error more than once, APPINITE switches to more detailed spoken prompts that ask the user to refer to contents and properties shown on the screen about the target UI object of the demonstrated action when describing the intention in natural language. Our user study showed that this helped users give successful descriptions (see Section VI).

At the end of the demonstration, the resulting script will be stored, and can later be invoked either using a GUI, from an external web service, by an IoT device or through an intelligent agent using the script execution mechanisms provided by the SUGILITE framework [2], [40]. The script can also be generalized (e.g., using a script demonstrated for making a reservation at a steakhouse to also make a reservation at a sushi restaurant) using script generalization mechanisms provided in SUGILITE [2].

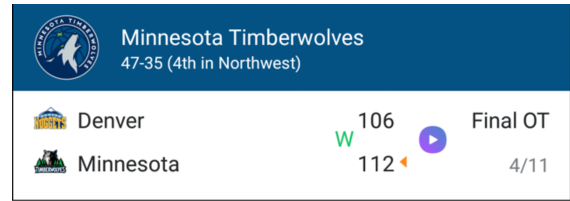
V. DESIGN AND IMPLEMENTATION

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.

A. 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 [41]




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

Fig. 4. 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.'"

similar to the DOM tree used in HTML. Each node in the tree is a *view*, representing a UI object that is visible (e.g., buttons, text views, images) or invisible (often created for layout purposes). Each view also contains properties such as its Java class name, app package name, coordinates for its on-screen bounding box, accessibility label (if any), and raw text string (if any). Unlike a DOM, our extracted hierarchical tree does not contain specifications for the GUI layout other than absolute coordinates at the time of extraction. It does not contain any programming logic or meta-data for the text values in views, but only raw strings from the presentation layer. This hierarchical model is not adequate for our 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. Fig. 3 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 our 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 easily plug our 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. The hierarchical relations are still preserved in the graph, but converted into hasChild and hasParent relationships between the corresponding view entities. Properties (e.g., coordinates, text labels, class names) are also converted into relations, where the values of the properties are represented as entities. Two or more constants with the same value (e.g., two views with the same class name) are consolidated as a single constant entity connected to multiple view entities, allowing easy querying for views with shared properties values.

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

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

B. 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. Fig. 1e, Fig. 3 and Fig. 4 show some example queries.

Representing UI elements as a knowledge 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 our language as an *S*-expression that is composed of expressions that can be of three types: *joins, conjunctions* and *superlatives*, constructed by the following 7 grammar rules:

$$E \rightarrow e;\ E \rightarrow S;\ \ S \rightarrow (join\ r\ E);\ S \rightarrow (and\ S\ S)$$
$$T \rightarrow (ARG_MAX\ r\ S); T \rightarrow (ARG_MIN\ r\ S);\ Q \rightarrow S \mid T$$

where Q is the root non-terminal of the query expression, e is a terminal that represents a UI object entity, r is a terminal that represents a relation, and the rest of the non-terminals are used for intermediate derivations. Our language forms a subset of a more general formalism known as Lambda Dependency-based Compositional Semantics [47] a notationally simpler alternative to lambda calculus which is particularly well-suited for expressing queries over knowledge-graphs.

Our parser uses a Floating Parser architecture [48] 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 [49].

C. Interaction Proxy Overlay

Prior mobile app GUI-based PBD systems such as SUGILITE [2] instrument 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. 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 [5], [50], [51]). 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.

To address these issues, we implemented an interaction proxy [52] to add an interactive overlay on top of third-party GUIs. Our mechanism 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 (Fig. 1d), or the wrong object (Fig. 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.

VI. USER STUDY

We conducted a preliminary 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.

A. 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 our target users, this was not a goal of this study. The primary goal was to evaluate the usability of our interaction design on combining natural language instructions and demonstrations. The demonstration part of this usability study was based on SUGILITE's [2], which found no significant difference in PBD task performances among groups with different programming expertise. Our formative study (Section III) showed that non-programmers were able to provide adequate natural language instructions from which APPINITE can generate generalizable data descriptions.

B. 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., Fig. 1c and Fig. 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).

C. 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 the supplemental video figure 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.

D. 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 (1) having bigger training datasets with better coverage for words and expressions users may use in instructions, and (2) enabling better semantic understanding of GUIs (details in Section VII). 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 our 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).

VII. Discussion and Future Work

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 run offline performance evaluations for the UI snapshot graph extractor and the natural language instruction parser, and in-situ field studies to evaluate APPINITE's usage and performance for organic tasks in real-world settings.

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 is to leverage the GUI as a shared grounding for EUD. By asking users to describe intentions in natural language referring to GUI contents, our tool 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."

While APPINITE has already offered some semantic-based features to provide greater expressiveness than existing end user PBD task automation tools, participants were hoping for more powerful support to enable them to naturally express more complicated logic in a more flexible way. To achieve this, we plan to improve APPINITE in the following areas:

A. Learning Conceptual Knowledge

We plan to leverage recent advances in natural language processing (e.g., [53]) to enable APPINITE to learn new concepts from users through verbal instructions. More specifically, we want to support users to add new relations into UI snapshot graphs through conversations. For example, for the interface shown in Fig. 1a, users can currently say, "*the restaurant that is 804 feet away*" (corresponds to the `hasText` relation) or "*the closest restaurant*" (corresponds to the `containsDistance` relation), but not "*restaurants within walking distance*" as APPINITE does not yet know the concept of "walking distance." We plan to enable future versions of APPINITE to ask users to explain unknown (and possibly personalized) concepts. For this example, a user may say "*Walking distance means less than half a mile*", from which APPINITE can define a relation extractor for the `isWalkingDistance` modifier for existing objects with the `containsDistance` relation, and subsequently allow use of the new concept "walking distance" in future instructions.

B. Computation in Natural Language Instructions

Currently in APPINITE, users have a limited capability of specifying computations and comparisons in natural language instructions. For example, for the interface shown in Fig. 2b, users cannot use expressions like "*flights that are cheaper than $700*" or "*if the flight is shorter than 4 hours*" in specifying data descriptions, although the UI snapshot graph already contains the prices and the durations for all flights. Furthermore, users are not able to create control structures (e.g., conditionals, iterations, triggers) which would require computations and comparisons. To address this issue, we plan to leverage prior work on natural language programming [32], and more importantly, how non-programmers can naturally describe computations, control structures and logic in solutions to programming problems [54] to extend our parser so that it can understand naturally expressed computations and the corresponding control structures. However, even with advanced semantic parsing and natural language processing techniques, GUI demonstrations will still remain essential for grounding users' natural language inputs and resolving ambiguities in the natural language.

C. Better Semantic Understanding of GUIs

Future versions of APPINITE can benefit from having better semantic understanding of GUIs. Some understanding can be acquired from user instructions, while others can come from existing resources. As discussed previously, the format of our UI snapshot graph allows easy integration with existing knowledge bases, which enables APPINITE to understand the semantics of entities (e.g., JetBlue, Delta and American are all instances of *airlines* for the interface in Fig. 2b). This integration can allow APPINITE to have more accurate instruction parsing, and to ask more specific questions in follow-up conversations.

GUI layouts can also be better leveraged to extract semantics. So far, we have only used the inter-object binary geometric relations such as `above` and `nextTo` to represent possible semantic relations between individual UI objects, but not the overall layout. Prior research suggests that app GUI designs often follow common design patterns, where the layout can suggest its functionality [55]. Also, for graphics in GUIs, especially for those without developer-provided accessibility labels, we can use runtime annotation techniques [56] to annotate their meanings. Visual features in GUIs can also be used in data descriptions, as discussed in [6], [57], [58].

VIII. Conclusion

Natural language instruction is a natural and expressive medium for users to specify their intentions and can provide useful complementary information about user intentions when used in conjunction with other EUD approaches, such as PBD. APPINITE combines natural language instructions with demonstrations to provide end users with greater expressive power to create more generalized GUI automation scripts, while retaining usability, transparency and understandability.

REFERENCES

[1] T. J.-J. Li, I. Labutov, B. A. Myers, A. Azaria, A. I. Rudnicky, and T. M. Mitchell, “An End User Development Approach for Failure Handling in Goal-oriented Conversational Agents,” in *Studies in Conversational UX Design*, Springer, 2018.

[2] T. J.-J. Li, A. Azaria, and B. A. Myers, “SUGILITE: Creating Multimodal Smartphone Automation by Demonstration,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2017, pp. 6038–6049.

[3] J. Allen *et al.*, “Plow: A collaborative task learning agent,” in *Proceedings of the National Conference on Artificial Intelligence*, 2007, vol. 22, p. 1514.

[4] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, “CoScripter: Automating & Sharing How-to Knowledge in the Enterprise,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2008, pp. 1719–1728.

[5] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller, “Automation and customization of rendered web pages,” in *Proceedings of the 18th annual ACM symposium on User interface software and technology*, 2005, pp. 163–172.

[6] T. Yeh, T.-H. Chang, and R. C. Miller, “Sikuli: Using GUI Screenshots for Search and Automation,” in *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2009, pp. 183–192.

[7] “Automate: everyday automation for Android. LlamaLab.” [Online]. Available: http://llamalab.com/automate/. [Accessed: 11-Sep-2016].

[8] S. K. Kuttal, A. Sarma, and G. Rothermel, “History repeats itself more easily when you log it: Versioning for mashups,” in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 69–72.

[9] M. Pruett, *Yahoo! Pipes*, First. O’Reilly, 2007.

[10] A. Cypher and D. C. Halbert, *Watch what I do: programming by demonstration*. MIT press, 1993.

[11] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[12] D. C. Halbert, “SmallStar: programming by demonstration in the desktop metaphor,” in *Watch what I do*, 1993, pp. 103–123.

[13] B. A. Myers and R. McDaniel, “Sometimes you need a little intelligence, sometimes you need a lot,” *Your Wish My Command Program. Ex. San Franc. CA Morgan Kaufmann Publ.*, pp. 45–60, 2001.

[14] B. A. Myers, “Scripting graphical applications by demonstration,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1998, pp. 534–541.

[15] B. A. Myers, “Peridot: creating user interfaces by demonstration,” in *Watch what I do*, 1993, pp. 125–153.

[16] S. Gulwani, “Automating String Processing in Spreadsheets Using Input-output Examples,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2011, pp. 317–330.

[17] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, “Programming by Demonstration Using Version Space Algebra,” *Mach Learn*, vol. 53, no. 1–2, pp. 111–156, Oct. 2003.

[18] T. J.-J. Li and O. Riva, “KITE: Building conversational bots from mobile apps,” in *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2018)*, 2018.

[19] R. G. McDaniel and B. A. Myers, “Getting More out of Programming-by-demonstration,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 1999, pp. 442–449.

[20] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, “A Machine Learning Framework for Programming by Example,” presented at the Proceedings of the 30th International Conference on Machine Learning (ICML-13), 2013, pp. 187–195.

[21] T. Lau, “Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI,” *AI Mag.*, vol. 30, no. 4, pp. 65–67, Oct. 2009.

[22] T. Y. Lee, C. Dugan, and B. B. Bederson, “Towards Understanding Human Mistakes of Programming by Example: An Online User Study,” in *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, New York, NY, USA, 2017, pp. 257–261.

[23] S. Oviatt, “Ten Myths of Multimodal Interaction,” *Commun ACM*, vol. 42, no. 11, pp. 74–81, Nov. 1999.

[24] R. A. Bolt, “‘Put-that-there’: Voice and Gesture at the Graphics Interface,” in *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1980, pp. 262–270.

[25] R. Marin, P. J. Sanz, P. Nebot, and R. Wirz, “A multimodal interface to control a robot arm via the web: a case study on remote programming,” *IEEE Trans. Ind. Electron.*, vol. 52, no. 6, pp. 1506–1520, Dec. 2005.

[26] S. Iba, C. J. J. Paredis, and P. K. Khosla, “Interactive Multimodal Robot Programming,” *Int. J. Robot. Res.*, vol. 24, no. 1, pp. 83–104, Jan. 2005.

[27] S. Oviatt, “Mutual disambiguation of recognition errors in a multimodel architecture,” in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1999, pp. 576–583.

[28] E. Adar, M. Dontcheva, and G. Laput, “CommandSpace: Modeling the Relationships Between Tasks, Descriptions and Features,” in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2014, pp. 167–176.

[29] T. Kasturi *et al.*, “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*, 2015, pp. 441–443.

[30] G. P. Laput *et al.*, “PixelTone: A Multimodal Interface for Image Editing,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2013, pp. 2185–2194.

[31] A. Azaria, J. Krishnamurthy, and T. M. Mitchell, “Instructable Intelligent Personal Agent,” in *Proc. The 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2016, vol. 4.

[32] A. W. Biermann, “Natural Language Programming,” in *Computer Program Synthesis Methodologies*, Springer, Dordrecht, 1983, pp. 335–368.

[33] D. Price, E. Rilofff, J. Zachary, and B. Harvey, “NaturalJava: A Natural Language Interface for Programming in Java,” in *Proceedings of the 5th International Conference on Intelligent User Interfaces*, New York, NY, USA, 2000, pp. 207–211.

[34] A. Begel and S. L. Graham, “Spoken programs,” in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05)*, 2005, pp. 99–106.

[35] H. Lieberman and H. Liu, “Feasibility studies for programming in natural language,” in *End User Development*, Springer, 2006, pp. 459–473.

[36] D. L. Chen and R. J. Mooney, “Learning to Interpret Natural Language Navigation Instructions from Observations,” in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, San Francisco, California, 2011, pp. 859–865.

[37] J. Thomason, S. Zhang, R. Mooney, and P. Stone, “Learning to Interpret Natural Language Commands Through Human-robot Dialog,” in *Proceedings of the 24th International Conference on Artificial Intelligence*, Buenos Aires, Argentina, 2015, pp. 1923–1929.

[38] V. Le, S. Gulwani, and Z. Su, “SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language,” in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA, 2013, pp. 193–206.

[39] C. Huff and D. Tingley, “‘Who are these people?’ Evaluating the demographic characteristics and political preferences of MTurk survey respondents,” *Res. Polit.*, vol. 2, no. 3, p. 2053168015604648, 2015.

[40] T. J.-J. Li, Y. Li, F. Chen, and B. A. Myers, “Programming IoT Devices by Demonstration Using Mobile Apps,” in *End-User Development*, Cham, 2017, pp. 3–17.

[41] Google, “AccessibilityWindowInfo | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityWindowInfo.html. [Accessed: 23-Apr-2018].

[42] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “Dbpedia: A nucleus for a web of open data,” *Semantic Web*, pp. 722–735, 2007.

[43] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, “Freebase: a collaboratively created graph database for structuring human knowledge,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1247–1250.

[44] D. Vrandečić and M. Krötzsch, “Wikidata: a free collaborative knowledgebase,” *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[45] S. Sen, T. J.-J. Li, WikiBrain Team, and B. Hecht, “WikiBrain: Democratizing computation on Wikipedia,” in *Proceedings of the 10th International Symposium on Open Collaboration (WikiSym + OpenSym 2014)*, 2014.

[46] T. J.-J. Li, S. Sen, and B. Hecht, “Leveraging Advances in Natural Language Processing to Better Understand Tobler’s First Law of Geography,” in *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA, 2014, pp. 513–516.

[47] P. Liang, M. I. Jordan, and D. Klein, “Learning dependency-based compositional semantics,” *Comput. Linguist.*, vol. 39, no. 2, pp. 389–446, 2013.

[48] P. Pasupat and P. Liang, “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*, 2015.

[49] J. Berant, A. Chou, R. Frostig, and P. Liang, “Semantic parsing on freebase from question-answer pairs,” in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 2013, pp. 1533–1544.

[50] M. Toomim, S. M. Drucker, M. Dontcheva, A. Rahimi, B. Thomson, and J. A. Landay, “Attaching UI Enhancements to Websites with End Users,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2009, pp. 1859–1868.

[51] J. R. Eagan, M. Beaudouin-Lafon, and W. E. Mackay, “Cracking the Cocoa Nut: User Interface Programming at Runtime,” in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2011, pp. 225–234.

[52] X. Zhang, A. S. Ross, A. Caspi, J. Fogarty, and J. O. Wobbrock, “Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2017, pp. 6024–6037.

[53] S. Srivastava, I. Labutov, and T. Mitchell, “Joint concept learning and semantic parsing from natural language explanations,” in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 1527–1536.

[54] J. F. Pane, B. A. Myers, and others, “Studying the language and structure in non-programmers’ solutions to programming problems,” *Int. J. Hum.-Comput. Stud.*, vol. 54, no. 2, pp. 237–264, 2001.

[55] B. Deka, Z. Huang, and R. Kumar, “ERICA: Interaction Mining Mobile Apps,” in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, New York, NY, USA, 2016, pp. 767–776.

[56] X. Zhang, A. S. Ross, and J. Fogarty, “Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement,” in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018.

[57] T. Intharah, D. Turmukhambetov, and G. J. Brostow, “Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions,” in *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, New York, NY, USA, 2017, pp. 233–243.

[58] M. Dixon and J. Fogarty, “Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2010, pp. 1525–1534.