SUGILITE: Creating Multimodal Smartphone Automation by Demonstration

Toby Jia-Jun Li[1], Amos Azaria[2], and Brad A. Myers[1]
[1] Human-Computer Interaction Institute, Carnegie Mellon University
[2] Computer Science Department, Ariel University
{tobyli, bam}@cs.cmu.edu, amos.azaria@ariel.ac.il

ABSTRACT
SUGILITE is a new programming-by-demonstration (PBD) system that enables users to create automation on smartphones. SUGILITE uses Android's accessibility API to support automating arbitrary tasks in any Android app (or even across multiple apps). When the user gives verbal commands that SUGILITE does not know how to execute, the user can demonstrate by directly manipulating the regular apps' user interface. By leveraging the verbal instructions, the demonstrated procedures, and the apps' UI hierarchy structures, SUGILITE can automatically generalize the script from the recorded actions, so SUGILITE learns how to perform tasks with different variations and parameters from a single demonstration. Extensive error handling and context checking support forking the script when new situations are encountered, and provide robustness if the apps change their user interface. Our lab study suggests that users with little or no programming knowledge can successfully automate smartphone tasks using SUGILITE.

Author Keywords
Programming by demonstration; smartphone automation; end-user development.

ACM Classification Keywords
H.5.2. Information interfaces and presentation (e.g., HCI): User Interfaces - *Interaction styles.*

INTRODUCTION
In smartphone usage, many common tasks are repetitive and tedious. Some also require a large number of actions, or navigating through a complex user interface (UI) structure. For instance, the current version of the Starbucks app for Android requires 18 taps to order a venti Iced Cappuccino with skim milk, and even more if the user does not have the account information stored. For those tasks, the users would often like to have them automated [2,26,38], for example, to have them performed by an intelligent software agent on their behalf [18]. In a motivating survey we conducted with 65 participants, 62.7% reported that they were interested in having a way to automate their repetitive tasks.

Intelligent agents like Siri, Cortana and Google Now can be activated by voice commands to perform various tasks, including device control, communication, web search, and calendar management. Such agents allow the user to focus on the specifications of the task while the agent performs the low-level actions [25], as opposed to the usual direct manipulation UI, in which the user must select the correct objects, execute the correct operations and control the environment [40]. However, prevailing smartphone intelligent software agents have limited functionality. They can only invoke built-in apps (e.g. Phone, Message, Calendar, Music etc.) and a few integrated external apps and web services (e.g. Search, Weather, Wikipedia). They lack the capability of controlling arbitrary third-party apps and services.

Companies like Apple are opening up the APIs to enable the third-party apps to integrate into their agents and to be activated from voice commands [45]. However, due to the cost and effort required, it is most likely that only some apps will be integrated and only some of the most popular tasks in those apps will be supported. Even for the supported apps, end users will still not be able to create automations for their personalized tasks and to integrate their personal requirements into the current smartphone agents. For example, most existing agents can invoke the music player, but they cannot incorporate the user's personal preferences in the automation, like applying a specific equalizer for an artist, or setting the volume to a pre-specified level before playing. Enabling personalization and customization is particularly important for modern smartphone users because they have very diverse patterns in what apps they use and how they use them [44].

Programming by Demonstration (PBD) is a promising technique to enable 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 [11,24,30]. This makes PBD particularly appealing for many smartphone users, who have little knowledge of conventional programming languages, but are familiar with how to perform the tasks they wish to automate using the existing smartphone apps.

CHI 2017, May 06 - 11, 2017, Denver, CO, USA.

ACM 978-1-4503-4655-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3025453.3025483

In this paper, we describe the design and implementation of SUGILITE[1], a PBD system that enables users to create task automations on smartphones and perform the tasks through a conversational interface. SUGILITE allows users to create generalized automation for tasks in arbitrary third-party mobile apps by simply demonstrating the procedure of performing the task using the regular app UI. 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. Our approach addresses the following three problems and limitations of other approaches:

Applicability: Many existing PBD systems [5,9,26] and mobile end-user development (EUD) systems [16,17] require the apps involved in the tasks to either use a special framework or library, or to provide an open API for their functionality. This limits the applicability of those systems – they can only be used to automate a small subset of tasks.

In contrast, SUGILITE can automate tasks using *any* third-party Android app (with a few exceptions noted in the Technical Limitation section), and can even create scripts that automate tasks across *multiple* apps. This makes PBD on smartphones much more useful, given nowadays an average U.S. smartphone user spends 90% of their mobile device time on apps [20], and uses 26.7 different apps [37].

Generalizability: Some previous macro-recording tools [39] do not suffer the applicability issues above. They can record a sequence of input events and replay the same actions at a later time. However, these scripts cannot be generalized, and will only perform the exact same tasks but not tasks with variations or different parameters. Other PBD systems (e.g. [27,32]) support generalization, but require multiple examples with different values for the parameters from the user. Prior studies have shown that end users often have a hard time giving meaningfully different examples for script generalization [12,28].

Our system has a multi-modal interface where the user can give a verbal command to execute an automation through a voice conversational interface, while making demonstrations and editing existing scripts using direct manipulation. In the background, SUGILITE detects the apps' UI hierarchy structures for all the activities that users visit. Then, SUGILITE combines the voice command, the actions recorded and an analysis of the app's structures to infer generalizations of the script. This allows SUGILITE to learn a generalized task from a single demonstration. In addition, if the recorded script encounters a new situation at runtime, then SUGILITE will allow the user to demonstrate new steps, which can either replace the original script (e.g., if the app has permanently changed the UI, or the users change what they want to do), or else can form a *fork*, and further generalize the script to work in the new situation. Finally, advanced users can manually generalize, review and edit the script through SUGILITE's editing interface.

Usability: Some mobile EUD systems (e.g. [38,46]) require users to program automations using a visual programming language or a textual scripting language. This imposes a significant learning barrier and prevents users with limited programming knowledge from using these systems.

In contrast, SUGILITE users can create automations by demonstrating the procedure using the familiar UIs of the actual apps involved. In our lab study, most participants were able to successfully automate tasks through SUGILITE, regardless of their prior programming experience.

Contributions

To summarize, the contributions of this paper are:

- SUGILITE, a mobile PBD system that enables the user to create an automation for arbitrary tasks across any or multiple third-party smartphone apps and to execute automated tasks through a multi-modal (speech) interface.
- A PBD script generalization mechanism that leverages the verbal command, the recorded actions, and recorded information about the UI hierarchy structures of the third-party apps to create a generalized program from a single demonstration. The system also has a representation of the recordings that allows users to manually edit scripts if necessary.
- 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.
- A lab study that showed users with different levels of programming experience were able to use our tool to create automation for four tasks derived from real-world scenarios with an 85.5% completion rate, plus subjective feedback showing they like and would want to use a tool like SUGILITE.

RELATED WORK

In this section, we discuss prior work in four areas related to our system: PBD in general, PBD generalization, instructable intelligent agents, and mobile end-user development (EUD) systems that use techniques other than PBD.

Task Automating by Demonstration

There have been many PBD systems to help people automate tasks [11,24]. 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 [9] only works with interface generated by the SUPPLE framework [14], and DOC-WIZARDS [5] only records actions performed on SWT widgets within Eclipse. Some PBD system focus on automating a specific type of task like file manipulation [29], photo manipulation [15], web tasks [23], or interactive interface construction [13,27]. Much work also has been done on PBD for human-robot interaction (e.g. [3,6,35])

[1] The source code is available at http://www.toby.li/sugilite_repo

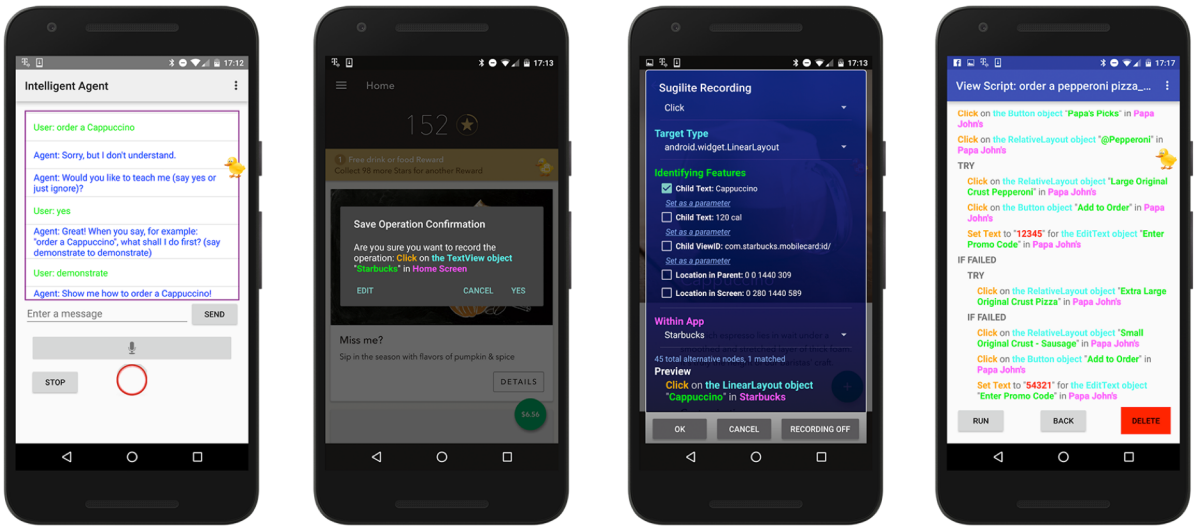

Figure 1. Screenshots of SUGILITE: (a) the conversational interface; (b) the recording confirmation popup; (c) the recording disambiguation/operation editing panel and (d) the viewing/editing script window.

Very few PBD systems exist on smartphones. KEEP DOING IT [26] derives IF-THEN automation rules from the user's demonstration. But it can only invoke system functions like turning on Wi-Fi, setting the ringer to silent, etc. It does not have the ability to control arbitrary third-party apps on the phone like SUGILITE does. To our knowledge, SUGILITE is the first PBD system to allow the automation of arbitrary tasks on any third-party app on a smartphone.

However, there are macro-recording tools on smartphone like [39] that can record a sequence of input events and replay them later. A major limitation of such tools compared to SUGILITE is that they 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.

Script Generalization

A major challenge for any PBD-based automation tool is script generalization [11,24,33] to support performing a *similar* task, but not necessarily the identical task [11]. Many approaches have been used in generalizing PBD scripts, including removing details (e.g. [19]), heuristics (e.g. [22,31,34]) and using multiple examples (e.g. [27]). SUGILITE uses a different approach, as discussed below.

Verbal Instruction

SUGILITE also uses verbal instructions in creating the automation. Prior works like [1,4,8,42] enable the users to instruct the agent to perform tasks using commands in natural language. A weakness of that approach on smartphone automation relates to efficiency – it can be slower to describe a smartphone operation in natural language than to simply tap on the screen. While we do not seek to make contributions in enabling better comprehension of verbal instructions, we leverage the current state-of-art work in comprehending verbal instructions to enhance the SUGILITE script generalizations.

End-User Development on Smartphone

Task automation is an important application for mobile EUD [36]. Besides PBD, there are other approaches to enable the end users to create automation on smartphones. Some approaches (e.g. [16,17,47]) use the "trigger-action" model, in which an "action" will be performed when the trigger happens. Compared with SUGILITE, these tools can only be applied on apps or services with open APIs available. The simple structures of their automation model also limit their uses in automating more complex tasks.

Alternatively, other systems like [38,46] allow the user to create automation with more complicated control structures (e.g. conditions, loops, exceptions). However, these suffer in usability compared with SUGILITE because they demand that users work with visual programming languages or textual scripting languages in unfamiliar interfaces.

EXAMPLE USAGE SCENARIO

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

Order Starbucks Coffee[2]

Smartphone users can order a wide range of products through the apps provided by merchants. In our 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 the same task is performed frequently.

[2] This scenario is also shown in the accompanying video.

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.

The user first gives SUGILITE a voice command, "Order a Cappuccino." using the conversational voice interface (Figure 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 also say verbal commands such as "Click on Starbucks", etc.) After each action, a confirmation dialog from SUGILITE pops up (Figure 1b) to confirm that the action has been recorded, 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 the Implementation section): "Cappuccino" and "120cal" (Figure 1c). When finished, the user clicks on the SUGILITE status icon and selects "End Recording".

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

This parameterization allows the user to give the voice command "Order a [DRINK]", where [DRINK] can be any of the drinks listed on the 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 can also be executed by using the SUGILITE graphical user interface (GUI) or invoked externally by a third party app using the SUGILITE API.

Additional Examples

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

- (American Express) "*Pay off my credit card balance."*
- (Venmo) *"Send [AMOUNT] dollars to [NAME]."*
- (Fly Delta) *"Find the flights from [CITY] to [CITY] on [DATE]."*
- (CHI 2016) *"Show me the papers by [NAME]."*
- (Transit) *"When is [BUS LINE] coming?"*
- (Pokémon Go) *"Collect my coins in the shop."*
- (GrubHub & Uber) "Request an Uber to the nearest *[TYPE]* restaurant

THE SUGILITE SYSTEM

Key Design Features

Multi-Modal Interface

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 the GUI. In order to create an automation, 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 when touching on the phone is not convenient, or for users with motor impairment.

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

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

The multi-modality of SUGILITE enables many useful usage scenarios in different contexts. For example, 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.

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.

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 UI 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.

Error Checking and Handling

Error checking and handling has been a major challenge for many PBD systems [21]. SUGILITE provides error handling and checking mechanism to detect when a new situation is encountered during execution or when the app's UI changes after an update.

When executing a script, an error occurs when the next operation in the script cannot be successfully performed. There are 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 cab during normal pricing, and then uses the script to request a cab 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 1d) should be used to enable the script to deal with a new situation. The forking works similarly to the try-catch statement in programming languages, where SUGILITE will first attempt to perform the original operation, and then execute the alternative branch if the original operation fails. (Other kinds of forks, branches, and conditions are planned as future work.)

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

Manual Script Editing

For advanced users, we provide an interface to manually edit and manually generalize the script (Figure 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 the Recording Handler section in Implementation 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.

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.

Conversational Interface

The conversational intelligent agent we used in SUGILITE is built using the Learning by Instruction Agent (LIA) [4]. When the user gives a voice command, the audio is first decoded into text using Google's Speech API. Then 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 [4].

Background Accessibility Service

The SUGILITE background service registers as an Android accessibility service. It listens to AccessibilityEvent[3] 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.

Recording Handler

The recording handler receives the AccessibilityEvent from the background service during the demonstration. 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.

[3] https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html

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. 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, SUGILITE seeks to find a unique second-level feature like view ID or label of a child element. If 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 (1) uniquely identify the UI layout of the screen and the specific target UI element on that screen, and (2) still work for future runs of the application to identify the same element.

If the handler determines that the heuristic-generated feature set can fulfill the above two requirements, the confirmation popup (Figure 1b) will be shown. Otherwise the disambiguation panel (Figure 1c) will be shown to ask the user to manually disambiguate what features should be used in the script. After this, SUGILITE generates and saves an operation to the current recorded script. The operation also stores identifiers for all the unique UI layouts and all the alternative UI elements that are structurally in parallel with the target UI element. After the demonstration, SUGILITE will use these when parameterizing and generalizing the script, as described earlier.

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.

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. Currently several research projects, both internal and external, are building upon or invoking SUGILITE.

Technical Limitations

Web-based Application: Because of the limitation of the Android accessibility mechanism, SUGILITE cannot record actions performed inside of browsers or in web-based applications. In the future, we can solve this by incorporating techniques used to access the HTML document object model (DOM) in existing web automation tools (e.g. [7,23]).

Graphical Icons: While many major apps are adding alternative texts to the icons to be more accessible, the icons in some popular apps remain unlabeled. For the icons with no text label or accessibility alternative text, the only way SUGILITE can identify them is by using the screen location, which is unreliable and not consistent across devices. For future work, we plan to use visual references similar to SIKULI [43] to identify the graphical elements in the script.

Gestures, Sensory Inputs and Text Entry: Due to the limitation of the Android accessibility mechanism, SUGILITE cannot record gestures or inputs from phone sensors (camera, accelerometer, gyroscope, etc.) during the demonstration. We can get access to the raw touch events and the raw sensor data with root permission of the phone, but that would lose SUGILITE's compatibility with unmodified phones. For the same reason, SUGILITE cannot record text typed directly into third-party apps using the on-screen keyboard. Instead, for text entry, users need to double tap on the textbox and type into a popup shown by SUGILITE.

Semantics in the Demonstration: In the current generalization mechanism of SUGILITE, we are treating the identifying features of elements and the parameters as strings, but SUGILITE does not attempt to understand their semantics or meaning. This means that in order to have proper generalization, the parameters in the user's verbal command have to match exactly the features of the target UI elements or the arguments of the operations. For example, if the user says "show me the flights to L.A. today" but demonstrates typing "Los Angeles" into the destination textbox in the Fly Delta app, then the script will not be automatically generalized. To address this, we are collaborating with researchers in natural language processing, who are working on understanding the semantics in verbal commands and smartphone apps [10,41]. This will enable SUGILITE to perform better script generalization from verbal commands.

EVALUATION

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. (These tasks focus on recording "straight-line" scripts, and future work will evaluate the understandability of error handling and other control structures.)

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. In Table 1, we show the description used for each category and the number of participants in each.

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
Total	19

Table 1. Number of participants (#) grouped by programming experience

Tasks

We chose five tasks for the study based on the common repetitive task scenarios from our 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 descriptions for each task, together with the SUGILITE components and commands used.

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.

Task1: 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.

Task2: 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.

Task3: Order Coffee

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

Task4: 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]"

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.

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 2. All times are in seconds.

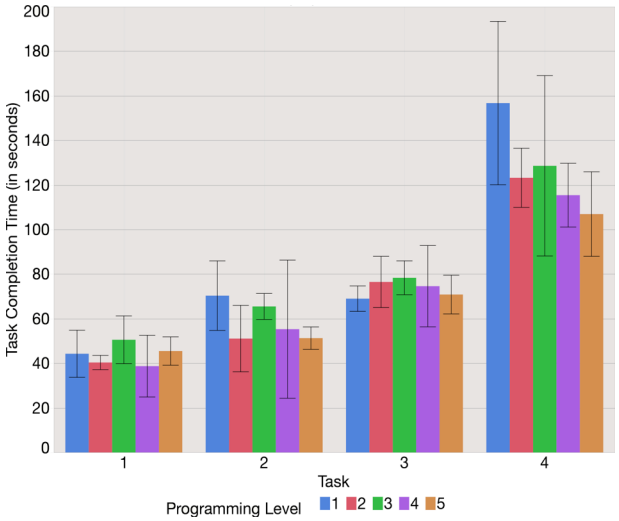

Figure 2. The average task completion time for the participants grouped by their programming experience. Shorter bars are better. Error bars show the standard deviations.

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 (they were supposed to double tap on the textbox and type into a pop-up due to the limitation discussed in the Technical Limitation section).

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.

Time Trade-off

For each of the four tasks, we also calculated an average "break-even" point at 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. 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 ($\overline{t_m}$) and the time it took to execute the automation (t_0), we calculate the average break-even value of n for each task, shown in Table 2.

Task	$\bar{t}$	$\overline{t_m}$	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 2. Average time to automate the task ($\bar{t}$), average time to perform the task manually ($\overline{t_m}$), time to run the automation (t_0), and the "break-even" point (n) for the four tasks.

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

Subjective Feedback

Overall, SUGILITE received positive feedback on both usability and usefulness from the 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 shows the average score for the usability-related items on the questionnaire. Table 4 shows the average score for the usefulness-related questions.

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. 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.

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 customized 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 although some apps have already been integrated in the prevailing agents like Siri, some specific tasks they want 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 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.

Besides the scenarios mentioned above, which SUGILITE is already capable of handling, participants also inspired us on potential directions for future work such as multi-device interaction, cross-user script sharing, smart device integration and enhanced accessibility support.

FUTURE WORK

From our experiences in the lab study, SUGILITE has been shown to be reliable and robust. For the next step, we plan to conduct a longitudinal field study of SUGILITE usage to help us understand how users use SUGILITE to help them on tasks in real life contexts. We plan to monitor what tasks they choose to automate, how they interact with SUGILITE in different modalities in different contexts, how they handle errors, and how much benefit SUGILITE can provide in real smartphone usage. Eventually we plan to release SUGILITE for general use by the public.

For future development, we plan to first solve the usability issues in text entry and the disambiguation panel raised in our lab study. We will develop a customized on-screen keyboard with keystroke recording to replace the default keyboard in the SUGILITE demonstration mode. This will enable SUGILITE to record what users type during the demonstration. We also plan to redesign the disambiguation panel to only display the most relevant information to avoid information overload. Finally, we hope to address the issues described in the Technical Limitation section and throughout this paper to expand the applicability of SUGILITE.

Two major concerns about our system are privacy and security. While we have not particularly focused on them in this paper, we do have plans for future work. In addition to the common practices like encryption and authentication, we are interested in detecting the "crucial steps" in a demonstration, such as the steps that are not undo-able (e.g. submit order, confirm deletion) or those that the user would like to review to ensure correctness. Once SUGILITE can successfully detect the crucial steps, it can then ask the user for confirmation by voice or in the GUI during execution.

CONCLUSION

Even though there have been many years of research on programming by demonstration and speech interfaces, SUGILITE is the first PBD system to show how they can successfully be put together on a smartphone to enhance the capabilities of both. We look forward to future collaborations between HCI and AI researchers for improved multimodal intelligent PBD user interfaces.

ACKNOWLEDGEMENTS

This work was supported in part by Yahoo! InMind project and by Samsung under GRO grant #A017479.

REFERENCES

1. James Allen, Nathanael Chambers, George Ferguson, et al. 2007. Plow: A collaborative task learning agent. In *Proceedings of the National Conference on Artificial Intelligence*, 1514.
2. V. Antila, J. Polet, A. Lämsä, and J. Liikka. 2012. RoutineMaker: Towards end-user automation of daily routines using smartphones. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 399–402. http://doi.org/10.1109/PerComW.2012.6197519
3. Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robotics and autonomous systems* 57, 5: 469–483.
4. Amos Azaria, Jayant Krishnamurthy, and Tom M. Mitchell. 2016. Instructable intelligent personal agent. In *Proc. The 30th AAAI Conference on Artificial Intelligence (AAAI)*.
5. 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), 191–200. http://doi.org/10.1145/1095034.1095067
6. Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. 2008. Robot programming by demonstration. In *Springer handbook of robotics*. Springer, 1371–1394.
7. 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*, 163–172.
8. David L. Chen and Raymond J. Mooney. 2011. Learning to Interpret Natural Language Navigation Instructions from Observations. In *AAAI*, 1–2.
9. 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), 159–168. http://doi.org/10.1145/1378773.1378794
10. Yun-Nung Chen, Ming Sun, and Alexander I. Rudnicky. 2015. Matrix factorization with domain knowledge and behavioral patterns for intent modeling. In *NIPS Workshop on Machine Learning for SLU and Interaction*.
11. Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.
12. Martin R. Frank and James D. Foley. 1993. Model-based User Interface Design by Example and by Interview. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology* (UIST '93), 129–137. http://doi.org/10.1145/168642.168655
13. 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), 95–101. http://doi.org/10.1145/192426.192466
14. Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces*, 93–100.
15. 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), 66:1–66:9. http://doi.org/10.1145/1576246.1531372
16. Ting-Hao Kenneth Huang, Amos Azaria, and Jeffrey P. Bigham. 2016. InstructableCrowd: Creating IF-THEN Rules via Conversations with the Crowd. 1555–1562. http://doi.org/10.1145/2851581.2892502
17. IFTTT. IFTTT. *IFTTT / Connect the apps you love.*
18. Jiepu Jiang, Ahmed Hassan Awadallah, Rosie Jones, et al. 2015. Automatic Online Evaluation of Intelligent Assistants. In *Proceedings of the 24th International Conference on World Wide Web* (WWW '15), 506–516. http://doi.org/10.1145/2736277.2741669
19. Ken Kahn. 1996. Toontalk TM—an animated programming environment for children. *Journal of Visual Languages & Computing* 7, 2: 197–217.
20. Simon Khalaf. Seven Years Into The Mobile Revolution: Content is King… Again. *Yahoo Developer Network*.
21. Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4: 65.
22. Tessa A. Lau and Daniel S. Weld. 1999. Programming by Demonstration: An Inductive Learning Formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces* (IUI '99), 145–152. http://doi.org/10.1145/291080.291104
23. 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), 1719–1728. http://doi.org/10.1145/1357054.1357323
24. Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.
25. Pattie Maes. 1994. Agents That Reduce Work and Information Overload. *Commun. ACM* 37, 7: 30–40. http://doi.org/10.1145/176789.176792
26. Rodrigo de A. Maués and Simone Diniz Junqueira Barbosa. 2013. Keep Doing What I Just Did: Automating Smartphones by Demonstration. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services* (MobileHCI '13), 295–303. http://doi.org/10.1145/2493190.2493216
27. 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), 81–82. http://doi.org/10.1145/263407.263515

28. 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), 442–449. http://doi.org/10.1145/302979.303127
29. 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), 455–456. http://doi.org/10.1145/259963.260464
30. 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), 59–66. http://doi.org/10.1145/22627.22349
31. Brad A. Myers. 1990. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 2: 143–177.
32. Brad A. Myers. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 243–249.
33. 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*: 45–60.
34. Brad A. Myers, Brad Vandcr Zanden, and Roger B. Dannenberg. 1989. Creating graphical interactive application objects by demonstration. In *Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology*, 95–104.
35. Shin 'ichiro Nakaoka, Atsushi Nakazawa, Fumio Kanehiro, et al. 2007. Learning from observation paradigm: Leg task models for enabling a biped humanoid robot to imitate human dances. *The International Journal of Robotics Research* 26, 8: 829–844.
36. A. Namoun, A. Daskalopoulou, N. Mehandjiev, and Z. Xun. 2016. Exploring Mobile End User Development: Existing Use and Design Factors. *IEEE Transactions on Software Engineering* PP, 99: 1–1. http://doi.org/10.1109/TSE.2016.2532873
37. Nielsen. 2015. So Many Apps, So Much More Time for Entertainment.
38. 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), 4:1–4:6. http://doi.org/10.1145/2162081.2162087
39. André Rodrigues. 2015. Breaking Barriers with Assistive Macros. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility* (ASSETS '15), 351–352. http://doi.org/10.1145/2700648.2811322
40. Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. 2016. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, Boston.
41. Ming Sun, Yun-Nung Chen, and Alexander I. Rudnicky. 2016. HELPR: A Framework to Break the Barrier across Domains in Spoken Dialog Systems. In *International Workshop on Spoken Dialog Systems*.
42. Jesse Thomason, Shiqi Zhang, Raymond Mooney, and Peter Stone. 2015. Learning to interpret natural language commands through human-robot dialog. In *Proceedings of the Twenty-Fourth international joint conference on Artificial Intelligence (IJCAI)*.
43. 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), 183–192. http://doi.org/10.1145/1622176.1622213
44. Sha Zhao, Julian Ramos, Jianrong Tao, et al. 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), 498–509. http://doi.org/10.1145/2971648.2971696
45. SiriKit - Apple Developer. https://developer.apple.com/sirikit/
46. Automate - everyday automation for Android. Llama-Lab. http://llamalab.com/automate/
47. Workato - Connect your apps. Automate your work. *Workato*. https://www.workato.com/