Voice enabling mobile applications with UIVoice

Ahmad Bisher Tarakji
Samsung Research America
a.tarakji@samsung.com

Jian Xu[*]
Stony Brook University
jianxu1@cs.stonybrook.edu

Juan A. Colmenares[†]
Samsung Research America
juan.col@samsung.com

Iqbal Mohomed
Samsung Research America
i.mohomed@samsung.com

ABSTRACT

Improvements in cloud-based speech recognition have led to an explosion in voice assistants, as bespoke devices in the home, cars, wearables or on smart phones. In this paper, we present UIVoice, through which we enable voice assistants (that heavily utilize the cloud) to dynamically interact with mobile applications running in the edge. We present a framework that can be used by third party developers to easily create Voice User Interfaces (VUIs) on top of existing applications. We demonstrate the feasibility of our approach through a prototype based on Android and Amazon Alexa, describe how we added voice to several popular applications and provide an initial performance evaluation. We also highlight research challenges that are relevant to the edge computing community.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing systems and tools**; *Interactive systems and tools*;

1 INTRODUCTION

Recently there has been great excitement on the possibilities offered by edge computing to support applications in saving bandwidth, providing low latency experiences to users, preserving user privacy, etc. In this paper, we describe a novel edge application — extending the capabilities of a voice assistant by performing actions on a user's personal mobile device. Modern voice assistants perform most automatic speech recognition (ASR) and natural language understanding (NLU) functions in the cloud. On the other hand, mobile devices contain significant personal state of users, that which spans across applications. By giving voice assistants access to the applications and state within mobile devices, we believe that an immense degree of personalization can be achieved. However, this leads to interesting architecture and latency considerations that will be of interest to the edge computing community.

[*]Work done during internship at Samsung Research America.
[†]Work done at Samsung Research America.

EdgeSys'18, June 10–15, 2018, Munich, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5837-8/18/06…$15.00
https://doi.org/10.1145/3213344.3213353

The key mechanism we introduce to tie voice agents with mobile applications is *User Interface Automation.* That is, we enable developers[1] to automate *UI interactions* that a human user would perform on a mobile app, and connect these to *voice interactions* that an end user would engage with on an agent. This combination lets us "voice enable" applications that were never designed for interface with voice agents. A novel facet of our approach is the ability to have a dialogue with the user in the face of ambiguities.

There are four contributions in this paper: (i) we present the UIVoice system for creating Voice User Interfaces (VUIs) on top of existing voice agents and mobile applications, (ii) we describe a framework that simplifies the task of creating VUIs, (iii) we describe our prototype system, discuss sample VUIs for several popular mobile apps and present a detailed evaluation, andfi nally, (iv) we highlight interesting research problems in edge computing that emanate from our work.

In this paper, we start by giving a background on voice assistants and how they currently interface with third-party applications (§2). In §3, we introduce our UIVoice system prototype that works with Amazon's Alexa family of voice agents and the Android OS. We use our prototype to create VUIs for several popular mobile apps, which are described and evaluated in §4. In §5, we discuss various important issues that emerge from our research. We also provide related work on automation techniques developed in industry and academia in §6, and the paper concludes in §7.

2 BACKGROUND ON VOICE ASSISTANTS

In recent years, voice-based interactions with computers have been embodied in the persona of voice assistants or agents, which can be bespoke devices in the home, integrated into cars and wearables or on smartphones. At present, there is no standard software architecture for voice assistants. However, there are two conceptual operations that occur in processing user utterances. First, the user's utterance goes through an automated speech recognition (ASR) module. The key output of this module is an attempted transcription of what the user said - text albeit still with speech recognition errors. Next, the recognized text goes through a Natural Language Understanding (NLU) module that attempts to extract the desired user intent. Advanced NLU/NLP may add more complex processing to the user's voice utterance, such as emotion recognition and speaker recognition. ASR and NLU systems that allow programmability, typically require the third-party developer to specify a set of user utterances, and how to extract commands (subsequently

[1]This can be any developer or even a power user - not necessarily the one who created the specific application(s) being automated.

referred to as *Intents*) and parameters (subsequently referred to as *Slots*) from the utterance. For instance "Call my Mom" would initiate a "call phone number" *Intent* with the *Slot* "Mom" of type "Contact" from which the assistant should extract the number to call. Of course, there may be multiple such applications active at any given point in time. Ultimately, the job of the speech processing system is to take the user utterance, extract the intent of the user request, and route the set of action and parameters to the appropriate VUI application. One important thing to note about existing voice assistants from a systems perspective is that the majority of ASR and Intent classification occurs in the cloud (a notable exception is wake word recognition). Some systems do provide a limited recognition vocabulary on-device (e.g. functions to set alarms or make a call). This is a fundamental problem because the size of the recognition vocabulary has an impact on the performance of the ASR system (and its resource requirements). The prevalence of cloud-based ASR and NLU leads to latency and other challenges that are relevant to the edge computing community.

A key challenge with existing SDKs from major commercial vendors is that they require mobile application developers to modify their code to support VUIs [2]. Given the large number of applications on major app stores, this is a significant barrier to the use of voice assistants on mobile phones. The approach proposed in this paper allows *unmodified* applications to be voice enabled.

Creators of VUI-based applications have always had to design good voice interactions regardless of the mechanism used to implement them. Designers utilize good-practices such as VUI patterns [12, 13] and Grice's maxims of conversation [10] as a guideline to closer approximate human expectations. However, the task of good dialog design is more an art than exact science [3, 9]. This is an important matter but not tackled by our work. In our system, the creator of the voice-enabled application uses the primitives of the voice agent to define an effective dialog with the end-user.

3 UIVOICE

In this section we present the UIVoice system overview and outline its two main steps.

3.1 System Overview

The UIVoice system has two distinct steps: a creation step and execution step. In the creation step, a developer specifies the VUI they wish to create and define a mapping to an existing mobile application. To specify the VUI, the developer must choose the voice interactions (i.e. utterances) that the end user will say to invoke actions on the mobile app. They also specify which parts of these utterances are slots. The mapping defines which interaction script should be run for a given utterance. The second step is execution, in which the system accepts a voice command from a user, resolves it into a specific GUI interaction script with thefi lled parameters, and then executes it on the mobile phone that contains the targeted application. In the course of performing UI interactions, there may be ambiguities that surface. A novel facet of our approach is the ability to have a dialogue with the user when there are multiple valid paths for an interaction script to continue. Figure 1 shows a high-level system design for execution of UIVoice. From a systems perspective, we highlight the following: the voice assistant (i.e. Alexa speaker)

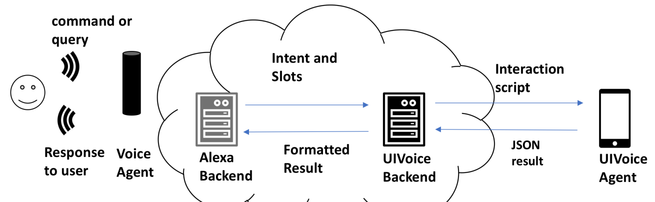

Figure 1: System design.

and mobile phone are connected to the home network. Since most of the ASR and NLU for Alexa occur in the cloud, this results in communication from the in-home device to the Alexa backend in the cloud. Logic in that tier connects with our UIVoice backend that is hosted in the cloud. Our UIVoice backend also maintains a communication channel to the mobile device running the UIVoice agent. This setup introduces various wide-area latencies that we discuss further during our evaluation.

The results of thefi rst step are two-fold: 1) code and configuration that needs to be uploaded to the voice agent system, and 2) instructions that are made ready to be sent to an agent on the mobile device. In the execution step, the UIVoice agent must be running on the user's phone, to enable remote interaction with the applications. In our implementation, we also have a *backend component* that runs on an external server and is responsible for mediating between the user's phone and the voice agent. In this section we describe all of these components.

3.2 VUI Creation Step

In this step, which occurs at development time, a VUI developer (this could be an end user who knows how to program or some third-party developer) specifies the voice interactions they wish to support and the mapping to an existing mobile application. These are both done by defining a *Interaction Script*. Conceptually, an interaction script has both user-facing actions (that execute in the context of the voice agent) and on-device actions. On the Voice agent side, actions are "listen" to the user (to capture the command the user wishes to execute for example) and "speak" to the user to "ask" them for input or "tell" them results. At the device side, actions might involve starting an application, navigating through multiple screens, reading parts of the screen, clicking UI elements, etc. Device-side actions are performed by the UIVoice device agent on the mobile device.

3.2.1 *Defining the Voice Interface.*

The voice interface is specified using the existing facilities of the voice agent. In our implementation, we focused on the Amazon Alexa, which provides the Alexa Skills Kit (ASK) framework. This framework enables developers to create third-party applications that run on a variety of devices such as the Echo and the Echo Show. Defining the voice interface requires the VUI developer to specify *Intents*, *Slots*, and *sample utterances*. The sample utterances help the NLP engine within Alexa to go from voice to a parsed set of intents and slots. Although these are specified in the interaction script, ASK does not allow dynamic uploading of skills. So currently we manually upload these to Alexa Skill Builder.

3.2.2 Defining Interaction Script on Mobile Device. This step is conceptually similar to creating a parametrized macro but on a mobile phone. In our implementation, we focused on Android in particular. The key mechanism that enables one to introspect and control what is displayed on screen is the Accessibility Framework [4]. Its original function was to allow users with impairments to productively use their Android device. It has also been used to do UI automation for testing [1]. By leveraging the Accessibility framework, we can specify actions that "find" certain UI elements and "interact" with these elements programmatically. [2]

To explore the possibilities and limits of creating these interaction scripts, we created an interactive tool in Python that let users type directives to query what is on screen as well as to perform actions on the results of these queries. This tool consists of a simple CLI that makes network requests to an agent running on the mobile device. The agent takes the action provided, executes it, and for queries, returns a result back to the caller. This tool is useful to understand the DOM structure of third-party applications and helps determine the precise action sequence to run in response to a command. As we worked with this interactive tool a pattern emerged. We would search for elements that matched certain criteria, select one of them, and then perform some user interaction on it. To this end, we identified a pattern called *Search-Select-Action*, which we describe next.

Search-Select-Action (SSA). As mentioned, the SSA pattern dominated most of the interaction scripts we explored. For example, to click the "Login" button in an app, the system would search all visible UI elements for buttons, select the button with the text (or description)fi eld "Login" and then perform the action "click" on it. We created all our scripts using a sequence of SSAs. The script is written in a declarative form, without need for explicit loops or intermediate variables.

For each UI interaction, a *Search* searches for a UI element in the application GUI. The GUI is represented as a "DOM-like" tree (similar to the familiar DOM-tree used to render web pages) and is constructed from data provided by Android's accessibility framework. The UI element does not need to be visible on-screen but needs to be loaded in the DOM-tree. The search may invoke one of many search strategies to narrow down the intended DOM elements. Each strategy will return a different set of results and can be configured by the developer to retrieve the UI elements she requires.

Most searches will return several results, which will need to be reduced to one result on which the action will be performed. This is done by a *Select*. Note that the Select phase could be merged with the Search phase as additional criteria when performing the query. Yet, we identified several cases in which the query would return several valid results and we needed the user to "select" one of them. In this case, the results are formatted and packaged into an appropriate response and sent back to the voice agent to "ask" the user for their selection. The execution on the mobile device will "pause" until it receives the user's selection and will resume after this ambiguity has been resolved. This is why we elected keeping the Select phase separate - to allow for this user interaction. The system will implicitly select thefi rst result if there is only one, or if the Select statement has been omitted completely.

[2]Android also provides a higher-level facility, called UIAutomation, to capture events from the accessibility system and enable operations on them. We used the lower-level accessibility events directly in order to have more control but did end up using some of the ideas from UIAutomator to deal with the high rate of events and to determine when the accessibility stream had stopped updating.

Finally, we have actions that can be invoked on the element selected. Not all elements accept every actions, so the system internally attempts to resolve errors and reverts back to the user in case it can not perform such an action. We support several actions such as *Launch App*, *Click*, *Scroll*, *Enter Text* and *Extract Text*. We also have a *Read* action that reads back to the user the result of last Search-Selection.

Final Interaction Script. To facilitate readability and ease of communication between voice agent and mobile phone, thefi nal interaction scripts are written in JSON. We defined several JSON objects that trigger Search, Select or Action methods on the UIVoice agent running on the phone. Examples of these JSON "commands" are shown later in the evaluation (Figure 3).

3.3 UIVoice Execution Step

After all scripts are built and uploaded, the system is ready to accept voice commands from the user and execute them on their mobile phones. Execution contains several components:

- *Voice Agent*: in our case Alexa. It takes the voice utterance from the user, matches it to the correct intent, extracts the variables (slots) from the utterance, and forwards the intent with the slots to the UIVoice backend server.
- *UIVoice Backend*: This is an AWS HTTP server that accepts requests from the voice agent with a certain Intent. It also allows the mobile phones to connect to it via WebSockets to facilitate quick and timely interactions. The main function of this component is to select the correct JSON interaction script based on the received intent and *parameterize* it with the slots received from the voice agent. It then forwards this JSON script to the phone for execution. Any *voice interactions* that might occur within the execution will also be mediated by this component.
- *UIVoice Agent on User's Smartphone*: This receives parametrized JSON and translates them into method calls on the UIVoice service. If a User-facing *Select* statement is invoked, it prompts the UIVoice backend to ask the user for additional input and then resumes execution with the selected option.

4 EVALUATION

In this section, we evaluate three popular mobile applications that we have voice-enabled using UIVoice (without modifying or recompiling them). Each app was supplemented with a useful function that users can perform via voice.

4.1 Test Applications

Here we briefly describe our test apps and their VUIs, in increasing order of complexity.

WhatsApp Messaging App. A common situation is when we need to tell someone we are running late. We created a simple VUI for WhatsApp that allows us to invoke the command "*Tell my partner [or husband, wife, etc.] I'm running late.*" This utterance makes the user's smartphone perform the following actions: 1) open WhatsApp, 2) select the recipient's name from the contacts and chats, 3) type out a well-formatted message, and 4)fi nally press the *Send* button.

YouTube Music Player. Popular music apps already have integrations built into voice assistants, but the integrations are very specific (*e.g.,* one can use a device to play a song by an artist or from a playlist). One of our current frustrations is with word collisions in artist names and song titles. For example, artists can have similar names, the same words may appear in song titles by different artists, and often the same song is covered by multiple artists. Music apps often give suggestions (via auto-completion) or options when the user types words in a search box. In our experience, we have not seen this type of functionality being provided through voice assistants in music apps.

Hence, we took YouTube Music and created a VUI that enables a user to say, "*Give me songs by [Artist].*" This vocal command results in the following actions on the user's mobile device: 1) start YouTube Music, 2) switch to and highlight the search box, 3) enter the name of the artist, as verbally given by the user, in the search box, 4) scrape thefi rst three suggestions given by the app, and 5) read out the artist names to the user. Besides artist names, we also have a variant for song titles. The VUI for this app differs from WhatsApp's VUI in two ways. First, we require an argument or "slot" from the user and second, we playback audio to the user.

LinkedIn Social News Reader. We created a VUI for the LinkedIn mobile app that allows the user to ask Alexa to fetch and read aloud a given number of news items from the user's personalized feed. The requested item count is a slot with a default value of 3, which is used when the user just asks Alexa for the news.

This VUI is interesting because, as of the date of submission, popular voice agents like Alexa and Google Assistant do not offer this functionality from the LinkedIn app. More importantly, it shows the efficacy of our approach – we were able to take an unmodified app from a third-party developer and created a specialized VUI for it using UIVoice. Note also that this VUI is more complex than the previous two as it involves scrolling a continuous feed. It highlights the benefits of the search-select-action scheme, which simplifies the interaction script needed to achieve the desired behavior.

4.2 Experimental Setup

The mobile device used in our evaluation is a Nexus 5 smartphone running Android Marshmallow. The UIVoice agent on the phone is written in Java (~3200 LOC). The Alexa device and the smartphone are connected to the Internet in Mountain View, CA. Alexa's ASR and NLU processing occur on Amazon's cloud. The UIVoice backend is written in Java (~350 LOC) and deployed in the `us-east` region of EC2 as required by Alexa.

4.3 Results

We start off by reporting the LOC for each interaction script as an indicator to the ease of creating these scripts. The LOC is 9, 6 and 6 for the interaction script for WhatsApp, Youtube Music and LinkedIn respectively. We also needed to specify 3-4 utterances for each, and specify the slot and slot type using primitives provided by

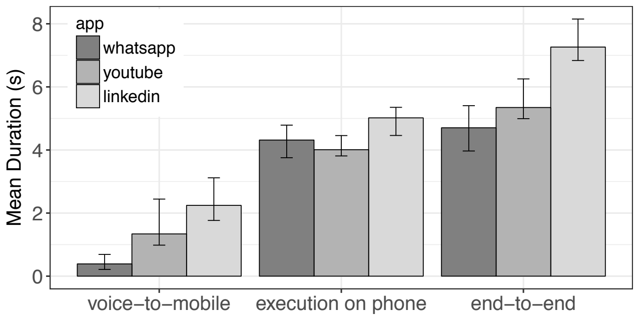

Figure 2: Average latency of the test apps over 10 runs. The error bars indicate min. and max. observed values.

Amazon Alexa. While the LOC count is low, writing each interaction script required about 2-3 hours.

Next we report the end-to-end response times of our test apps and the latency of UIVoice's cloud and smartphone components. Figure 2 shows our results. The left-most set of three bars indicate the elapsed time from when the userfi nishes speaking a phrase to when the corresponding call is received by the smartphone. We call this *voice–to–mobile* time. The elapsed time includes: 1) the network communication latency with Amazon's infrastructure, 2) the time for ASR and NLU in the Alexa system, and 3) the latency of UIVoice backend. We observed mean values around 1–2 seconds. We also observe that WhatsApp takes far less time than the other apps; we attribute this to WhatsApp's VUIfi lling no slots for NLU.

The bars in the middle of Figure 2 show the time it takes to process the VUI's logic on the smartphone. Note that a complex series of automated operations were performed in 4–5 seconds (more details are given below). Finally, the right-most bars in the figure report the overall processing time of user requests. The processing time is under 8 seconds in almost every case, which we consider reasonable for actual usage. However, a sophisticated VUI with a larger number of complex interactions on the device may take longer to process user requests. In this case, it would be prudent to keep the user informed about the progress in processing a request; it would prevent any misunderstanding that the system has not heard the user or has failed in some way. This is also important because Alexa times out interactions that take more than certain time t, currently $\sim$10 seconds.

Next, we discuss the detailed latency breakdown across interactive operations on the smartphone and compare them with the voice–to–mobile duration described above. We report several interesting results in Figure 3.

We observe that launching the apps take a significant amount of time when compared to other operations. While possibly acceptable in normal usage (when a user manually starts the apps), such delay is excessive for voice-based automated apps. Hiding the app launching delay is challenging for a system such as UIVoice because an interaction script must wait for the app to reach a well-known point from which to begin the automated interactions. Even though we do not explore this issue in our work, we believe the launching delay may be reduced through more engineering and research efforts.

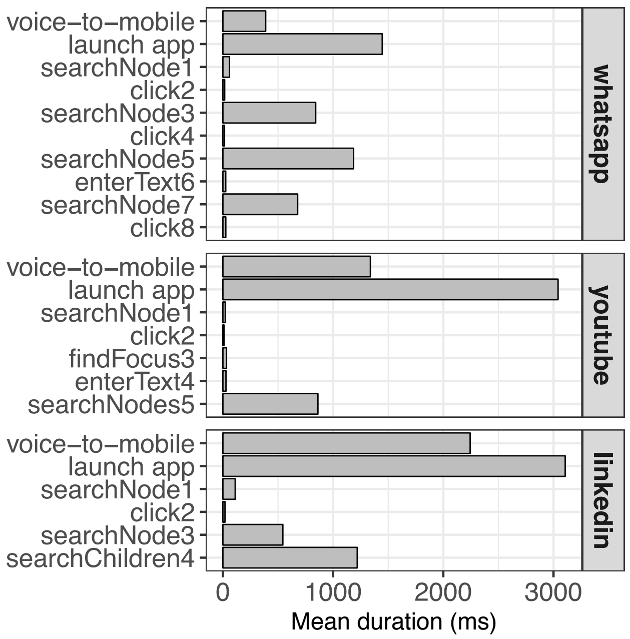

Figure 3: Average time taken by various automated interactions in the test apps over 10 runs.

Third, entering text and other typical interactive operations, such as clicking on UI elements, are very fast. This result suggests that UIVoice can save users time in repetitive tasks heavily involving typing and clicking. However, we observe that node search operations take much more time. Part of the time required is waiting for the graphical user interface to settle down after an interactive event has occurred. We use the same technique as Google's UIAutomator tool, and we specify afi xed delay called *quiesce time*. If the Accessibility event stream produces no changes for the specified amount of time, the UI is considered to have settled down. Through trial and error, we obtained the quiesce time we used in our evaluation (300 ms). If the value is too short, there is the danger that the UI does complete changing before the next automated interaction occurs, which is an error. On the other hand, if the value is too large, it slows down the operation of the interactive scripts. With automated interactions, this source of delay is worth reducing and likely achievable with more engineering effort.

Finally, we can see variability between the various `searchNode` and `searchNodes` operations in Figure 3. The reason is that in some cases we only search for a particular UI element (*e.g.,* a button) and the search through the accessibility DOM tree can be short-circuited, whereas in other cases we may have to go through the entire tree. Moreover, the `searchChildren` operation is particularly intensive as it may involve scrolling through screens, waiting for quiesce, and refreshing the accessibility DOM tree. Note that when the when the UI updates (*e.g.,* from a scroll interaction), most nodes in the DOM tree become invalid, and in our experience, to get an accurate "picture" of the UI after scrolling, we must retraverse the DOM tree.

5 FUTURE RESEARCH DIRECTIONS RELATED TO EDGE COMPUTING

Here, we list open challenges to the edge community.

Connecting the Home Edge to the cloud. Our research group has been exploring an edge computing platform for the home. One interesting aspect that emerges is how to connect devices in the home edge to the cloud (e.g. for voice-agent or mobile-device based control). Rather than have every device maintain connections to the cloud, it is more efficient for a single device to maintain that connection, and relay messages to other home devices. How to do this robustly and efficiently is an interesting research question.

Enable Multi-Device, Multi-Agent Operation. Voice agents have the potential to enable seamless multi-device interactions. By providing a consistent interface across multiple devices, one might consider them to be an OS for a user's home or their personal devices. Further, users often have their data (or inferences on their data) across different services, vendors, devices or intelligent agents. We see value in federation across intelligent agents. Our experience integrating Alexa with interactions on an Android phone suggests that such multi-device, multi-agent interactions may be feasible. Our belief is that the ideal mechanism would have less reliance on the cloud, and move towards P2P interactions between devices and assistants.

Low-latency actuations via Voice. Popular commercial voice agents today have significant reliance on the cloud. This proves challenging for voice-based device control when there is an expectation of low latency (*e.g.,* to control a fast-paced interactive game), and is an promising avenue of research.

6 RELATED WORK

Numerous commercial tools focus on automating tasks on smart phones and across devices. Examples include: Automate, MacroDroid, Tasker, IFTTT, and AutomateIt. These tools enable execution of a sequence of actions when a specified condition is met. UIVoice provides a declarative framework for creating voice and on-device interactions that might span multiple dialogs with the user and actions on the device.

Puppeteer [8] used OLE automation techniques to adapt contentfi delity of documents to resource availability. PageTailor [5] used automation in the browser to adapt UIs to small screens. UIWear [14] used android accessibility to enable companion apps on wearables. We use automation to enable creation of Voice UIs that leverage mobile apps.

Some recent work has considered how to facilitate interactions across device boundaries. Weave [7] provides a programming framework for developers that simplifies creating multi-device user interactions. Almond [6] provides a programming framework that allows intelligent agents to interface across devices. The novelty in our work is the use of UI automation to enable creation of VUIs using existing unmodified mobile applications.

SUGILITE [11] is a programming-by-demonstration system that lets users create automations on smartphones. It also supports parametrization of the automation and a limited form of verbal commands. This work is complementary to UIVoice, as it could enable end-users to create the UIVoice scripts by demonstrating the interactions on mobile app. UIVoice also identifies the *SSA* pattern that enables conversing with the user to choose options while executing the script, and our work also focuses on latency and other systems issues.

Samsung's Bixby agent 1.0 enables end-users to create *QuickCommands* and *Multi-QuickCommands* through voice. These "recorded" commands cannot be parametrized or involve dynamically loaded UI elements. In UIVoice, the sequence of interactions are created programmatically on top of unmodified mobile apps, and supports reading content back to the user as well as soliciting input in the course of an interaction. Our architecture also enables voice assistants not running on the phone to nonetheless interface with mobile apps.

7 CONCLUSIONS

In this paper, we introduced UIVoice, a system and framework that enables voice interactions for existing mobile applications without requiring any code modifications in the original application. As voice assistants have significant reliance on the cloud for their ASR and NLU functions, we believe our model which involves significant operations on end-user devices, poses interesting challenges for the edge computing community.

REFERENCES

[1] 2017. Accessibility Developer Guide for Android. https://developer.android.com/training/testing/ui-automator.html.

[2] 2017. Actions on Google Assistant. https://developers.google.com/actions/.

[3] 2017. Applying Built-in Hacks of Conversation to Your Voice UI (Google I/O '17). https://www.youtube.com/watch?v=wuDP_eygsvs.

[4] 2017. UIAutomator, a UI testing framework for Android. https://developer.android.com/guide/topics/ui/accessibility/index.html.

[5] Nilton Bila, Troy Ronda, Iqbal Mohomed, Khai N. Truong, and Eyal de Lara. 2007. PageTailor: Reusable End-user Customization for the Mobile Web. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services (MobiSys '07)*. ACM, New York, NY, USA, 16–29. https://doi.org/10.1145/1247660.1247666

[6] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant *(WWW '17)*. Geneva, Switzerland, 341–350. https://doi.org/10.1145/3038912.3052562

[7] Pei-Yu (Peggy) Chi and Yang Li. 2015. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3923–3932. https://doi.org/10.1145/2702123.2702451

[8] Eyal De Lara, Dan S. Wallach, and Willy Zwaenepoel. 2001. Puppeteer: Component-based Adaptation for Mobile Computing. In *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 3 (USITS'01)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=1251440.1251454

[9] R Frederking. 1996. GriceâĂŹs maxims: do the right thing. *Frederking, RE* (1996).

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

[11] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6038–6049. https://doi.org/10.1145/3025453.3025483

[12] Dirk Schnelle and Fernando Lyardet. 2006. Voice User Interface Design Patterns.. In *EuroPLoP*. 287–316.

[13] Dirk Schnelle, Fernando Lyardet, and Tao Wei. 2005. Audio Navigation Patterns. In *10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*. 237–260.

[14] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E. Porter. 2017. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom '17)*. ACM, New York, NY, USA, 369–382. https://doi.org/10.1145/3117811.3117819