Geno: A Developer Tool for Authoring Multimodal Interaction on Existing Web Applications

Ritam Jyoti Sarmah
UCLA HCI Research
rsarmah@g.ucla.edu

Yunpeng Ding
UCLA HCI Research
dyp1225@g.ucla.edu

Di Wang
UCSD Computer Science
diwang0503@gmail.com

Cheuk Yin Phipson Lee
UCLA HCI Research
phipsonleecy@gmail.com

Toby Jia-Jun Li
CMU HCII
tobyli@cs.cmu.edu

Xiang 'Anthony' Chen
UCLA HCI Research
xac@ucla.edu

ABSTRACT

Supporting voice commands in applications presents significant benefits to users. However, adding such support to existing GUI-based web apps is effort-consuming with a high learning barrier, as shown in our formative study, due to the lack of unified support for creating multimodal interfaces. We present Geno—a developer tool for adding the voice input modality to existing web apps without requiring significant NLP expertise. Geno provides a high-level workflow for developers to specify functionalities to be supported by voice (intents), create language models for detecting intents and the relevant information (parameters) from user utterances, and fulfill the intents by either programmatically invoking the corresponding functions or replaying GUI actions on the web app. Geno further supports multimodal references to GUI context in voice commands (*e.g.*, "move this [event] to next week" while pointing at an event with the cursor). In a study, developers with little NLP expertise were able to add multimodal voice command support for two existing web apps using Geno.

Author Keywords

Multimodal interaction; voice input; developer tool; natural language processing

CCS Concepts

•Human-centered computing → Human computer interaction (HCI);

INTRODUCTION

The advent of data-driven speech recognition and natural language processing (NLP) technology holds the promise of enabling robust and intelligent voice input that can recognize users' intent from natural language expression.

Meanwhile, as more applications become ubiquitously available on the web, adding multimodal, voice-enabled input on existing web apps presents important benefits. Voice input enhances GUI web apps' accessibility for visually-impaired users. Voice+GUI multimodal interaction also adds to the expressiveness of singular input modality [8, 17, 10, 33, 14].

However, currently it takes a significant amount of work to augment existing web apps to support voice+GUI input. Despite the availability of multiple existing APIs and toolkits (*e.g.*, Chrome, Mozilla, W3C, Annyang, Artyom), our formative studies with five web developers identified the following barriers: *(i)* the amount of new code to write, including the effort of refactoring the existing codebase; *(ii)* the gap of NLP expertise—to realize the NLP capability of a voice input, non-expert developers often find it laborious and challenging to develop mechansisms for understanding natural language inputs. *(iii)* the lack of unified, integrated support for creating multimodal interaction—developers find it hard to 'map' the development of voice input to their familiar GUI building paradigm and are unfamiliar with the best practice.

To ease the addition of voice input to GUI, prior research focused on enabling end-users to create custom voice assistants on personal devices, *e.g.*, smartphones [18, 21, 20]. While the end-user approach benefits users without significant technical expertise, it requires substantial efforts from *each* end user of the app, which does not easily scale up. To complement the end-user-oriented approach, we focus on developer tools that can help developers make voice input readily available on existing web apps for all end-users to "walk up and use". Further, beyond prior work that considered voice input as a separate modality, we want to support developers to integrate existing inputs (*e.g.*, mouse) multimodally with voice.

To achieve these goals, we develop Geno—a developer tool for authoring voice input single- or multimodally on existing web apps. Different from end-user tools that often revolve around demonstration at the front-end, Geno assumes that a developer is familiar with their own codebase at the back-end.

Scenario Walkthrough

Figure 1 shows an overview of the Geno IDE that consists of a file explorer, a code editor, and a preview of the web app.

Submitted to UIST 2020

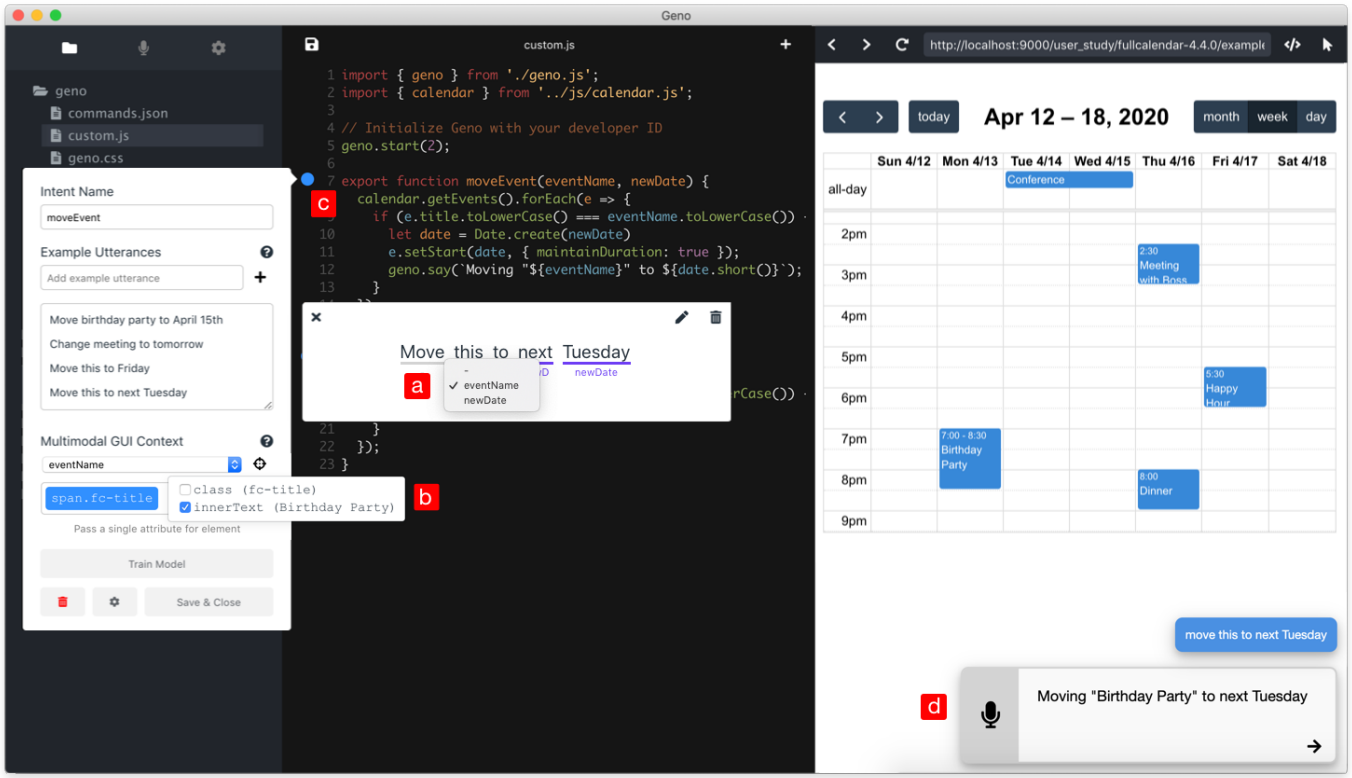

Figure 1. In a few steps, Geno enables a developer to create a multimodal voice+GUI input where a user can point at an calendar event and say when to move the event (d): #1 Specifying Target Action (*e.g.*, selecting the `moveEvent` function as the intent) (c); #2 Configuring Voice Input by providing and labeling example utterances (*e.g.*, "this" is the value for the `eventName` parameter) (a) #3 Adding GUI Context by demonstration (*e.g.*, hovering an event element where its `innerText` attribute will also be used for `eventName`) (b).

We demonstrate an exemplar workflow using Geno for a calendar app[1], where a specific 'view' often limits the direct manipulation of events. For instance, in the 'week view' (Figure 1), it is cumbersome to drag a calendar event to a different week or month. A developer wants to create a multimodal voice+GUI input to ease this interaction, *e.g.*, one can hover the mouse over an event and say "Move this to next Tuesday".

To achieve this task, Geno supports the developer to demonstrate possible multimodal inputs, identify relevant information from those inputs (called parameters), and use such parameters to programmatically invoke specific actions.

First, the developer provides an intent name and some sample utterances, and labels the parameter values in the utterances (Figure 1a). For example, as shown in Figure 1c, for the intent "moveEvent" the developer provides an utterance "Move this to next Tuesday", and labels the entities (extracted by Geno)—'this' as `eventName`, and 'next Tuesday' as `newDate`.

Next, the developer adds 'GUI context'—a non-voice modality—as an alternate means of specifying `eventName` by clicking any event element As shown in Figure 1c, Geno automatically identifies the clicked element and displays its attributes as a checklist. The developer checks the `innerText` property to be used as the value for `eventName`.

[1] https://fullcalendar.io/

Finally, the developer implements a simple wrapper function called `moveEvent`, whose skeleton is automatically generated in the current JavaScript file (Figure 1c) to make use of the two multimodal parameters extracted by Geno. Note that this step is specific to the internal logic of the developer's web app; by design, Geno maintains a loose coupling with such logic and does not attempt to automate this step.

At run-time, Geno floats over the calendar app (Figure 1d). Clicking the 🎙 icon (or using a keyboard shortcut) starts a multimodal voice+GUI input. Geno automatically recognizes the intent from the utterance, which leads to the target action of `moveEvent`. Geno will look for the two requisite parameters: `newDate` from the utterance and `eventName` from mouse events. If a parameter cannot be found, Geno will automatically prompt the user to manually specify the information.

Validation & Contributions

We conducted an hour-long developer study with eight participants, who were able to learn and use Geno to implement multimodal voice+GUI input on existing calendar and/or music player apps. Following Geno's high-level workflow, participants found it much easier to build voice/NLP-enabled interaction than they had previously expected. Meanwhile, it remained at times challenging to make certain design decisions (*e.g.*, to use function or GUI demonstration as the target action, which element/attribute to use as GUI context), which

could be better supported in the future with a comprehensive tutorial and support for *in situ* testing and visualization.

Overall, Geno makes a tool contribution to the development of multimodal input. Specifically,

- Geno embodies existing techniques for processing multimodal input while abstracting low-level details away from developers who might not be an expert;
- Geno unifies the creation of different modalities into a single workflow, thus lowering the complexity of programming an interaction multimodally;
- Geno works on existing web apps, whose accessibility can be enhanced by adding complementary input modalities;

RELATED WORK

We first review prior work on multimodal interaction techniques and applications and then discuss existing tool support for authoring multimodal interaction.

Multimodal Interaction Techniques & Applications

Multimodal interaction leverages the synergic combination of different modalities to optimize how users can accomplish interactive tasks [27]. One seminal work was 'Put-that-there', whereby a user can simply point at a virtual object and speak to the system to place that object by pointing at a location on a 2D projected display [2]. QuickSet demonstrated multimodal input on a military planning interface that allows a user to pen down on a map and utter the name of a unit (*e.g.*, "red T72 platoon") in order to place it at that location [8].

Prior work on the design of multimodal [36, 28] interaction motivates our second goal of combining an additional voice modality with existing GUI-based input: users can offload a subset of GUI-based tasks that can be more optimally accomplished via voice. To understand such complementary relationships between modalities. Oviatt found that users' natural language input is simplified during multimodal interaction, as the complexity is offloaded to more expressive modalities [26]. Cohen *et al.* pointed out that direct manipulation can effectively resolve anaphoric ambiguity and enable deixis commonly found in natural language communication [7].

Voice-enhanced, multimodal interaction has been explored in a plethora of application domains. Xspeak uses speech to manage windows in the XWindow System, such as saying the window names to switch between windows in focus [31]. Voice input can also ease tasks with a large number of GUI menu options such as image editing by directly speaking out the command [17]. On the other hand, when the content of the interaction is heavily loaded with information (*e.g.*, visualization of a complex data set), multimodal interaction allows the use of natural language to describe a query and direct manipulation to resolve ambiguities [10, 33]. The recent focus on automotive UI also suggests a promising scenario for introducing multimodal interaction. For example, SpeeT is a speech-enabled, multi-touch steering wheel that allows drivers to use speech to select a in-car functionality and gestures to adjust the settings [30]. Interacting with an Internet of Things can also benefit from a combination of freehand pointing and voice commands, as demonstrated in the Minuet system [14]. As Web-based UI becomes increasingly pervasive, it becomes important to offer flexible moddalities at the users' disposal, *e.g.*, using input targeting [35]; we expect our tool to be extensible to address multimodal interaction in these above application scenarios.

Tool Support for Authoring Multimodal Interaction

Previous research on tools for authoring multimodal interaction spans three approaches: architectural support, specification language, and end-user development.

Architectural support. The Open Interface Framework provides both a kernel and a development environment to allow for heterogenous, distributive interactive components of different modalities to compatibly run as an integral system [32, 22]. Bourguet separated the development of multimodal interaction into *(i)* specifying the interaction model using a finite state machine; and *(ii)* using a multimodal recognition engine to automatically detect events relevant to the interaction model [3, 4]. HephaisTK is a toolkit where recognition clients observe input from different modalities, which is integrated by fusion agents and can be subscribed by client applications [9]. mTalk is a multimodal browser that provides integrated support (*e.g.*, cloud-based speech recognition) to ease the development of multimodal mobile application [13]. Speechify is a toolkit that wraps speech recognition, speech synthesis, and voice activity detection APIs to assist with the rapid construction of speech-enabled Android apps [15]. Little of this work, however, supports the development of user interactions at the front end, which our research primarily addresses in Geno.

Specification language has been explored extensively in prior work. Obrenovic and Dusan described an UML-like language to represent multimodal interaction that can be generalized or transferred between different designs and analyses [25]. XISL is one of the earliest XML-based markup languages that focused on describing the synchronization of multimodal inputs/outputs, as well as dialog flow and transition [16]. MIML presents an important feature that creates three layers of abstraction when specifying multimodal interaction: task, interaction, and device [1]. UnisXML [34] and TERASA [29] take a similar transformational approach, *i.e.*, deriving an instance of multimodal interaction from abstractions of task, domain and user interfaces. Emma focused on capturing and annotating the various stages of processing of users' multimodal inputs [12]. Different from all this work, Geno provides an interactive authoring tool that overcomes the lack of directness and expressivity of using a specification language.

End-user development of multimodal interaction has become a promising option for end users to develop the support for their own desired tasks. Multimodality is often used to provide "naturalness" in the development process [24] to make it closer to the way the users think about the tasks [11]. Sugilite allows end-users to create voice-activated task automation by demonstrating the task via directly manipulating existing app GUIs [18]. Unlike Geno, Sugilite lets end users enable their personal tasks to be invoked by voice commands. Although

the "programming" process is multimodal, the invoking of automation uses only voice commands. In comparison, Geno is a developer tool that helps developers to quickly add the support for mutlimodal interactions for existing web GUIs. Pumice builds off of Sugilite and adds support for learning concepts in addition to procedures [20]. Improv allows end-users to replace direct manipulation with indirect gesturing on a second device, employing a programming by demonstration (PBD) approach to record and programmatically replay input events across devices to a web app [6]. Appinite uses a multimodal mixed-initiative mutual-disambiguation approach to help end-users clarify their intents for ambiguous demonstrated actions, addressing the challenge of "data description" in PBD [19].

Finally, there are commercial tool support, *e.g.*, Amazon Lex, which mainly focuses on the speech modality manifested as chat bots, rather than enabling multimodal interaction.

FORMATIVE STUDY

Following the approach for designing developer tools [23], we conducted a formative study to identify challenges in developing multimodal interaction and the gap of existing toolkits/libraries/APIs in their support of such development.

Participants

We recruited five participants (four male, one female, aged 19-23) from a local university majoring in Electrical Engineering and Computer Science (four undergraduate and one graduate students). All participants considered web development (JavaScript/HTML/CSS) as their skills, although their experiences varied, ranging from six months to two years. Only one participant reported having tried an API related to speech recognition and NLP, while others reported no related experience at all. Each participant received a $20 Amazon gift card for compensation.

Procedure & Tasks

Participants were randomly assigned to one of these two tasks of adding voice command capabilities to existing web applications based on existing open-sourced codebases:

- Task option 1–Calendar: create a voice command to add an event at a given date and time[2];
- Task option 2–Music player: create a voice command to play/pause music[3];

Participants were allowed to use any IDE and third-party toolkits/libraries/APIs; in case they could not find any, we also provided a list of five commonly used voice input related resources: W3C[4], Mozilla[5], Chrome[6], Artyom[7] and Annyang[8].

[2] https://github.com/fullcalendar/fullcalendar
[3] https://github.com/521dimensions/amplitudejs
[4] https://w3c.github.io/speech-api/
[5] https://developer.mozilla.org/en-US/docs/Web/API/SpeechRecognition
[6] https://developers.google.com/web/updates/2013/01/Voice-Driven-Web-Apps-Introduction-to-the-Web-Speech-API
[7] https://sdkcarlos.github.io/sites/artyom.html
[8] https://www.talater.com/annyang/

After a brief introduction, each participant was given 60 minutes to perform the development task, during which the experimenter asked them to think aloud. In the end, participants were asked to complete a short questionnaire to evaluate their overall experience and performance in the specified task.

Findings & Design Implications

All participants were unable to complete the assigned task of adding the support for voice input to a provided web app within 60 minutes. Admittedly, such an outcome could have been a result of two obstacles: a lack of web programming expertise in general and the unfamiliarity to voice input and NLP. Below we discuss our observations specific to the latter obstacle and the corresponding design implications for Geno.

(i) Due to a lack of NLP expertise, all participants struggled to come up with a programmatic way of parsing voice inputs. Most attempted to manually create 'hard-coded' rules that only worked for specific cases. Some APIs used by the participants (*e.g.*, W3C Speech Recognition API) only perform speech recognition but no parsing; some others (*e.g.*, Annyang) do provide parsing capabilities but requires knowledge of using a regular expression.
Design implication: the parsing of the transcribed voice input should be automatic and require as little extra knowledge and effort as possible.

(ii) Participants found the process of programming voice input unnatural and unfamiliar compared to their experience of developing non-voice GUI. Specifically, P5 suggests "defining a single uniform function" that could be reused across different modes of input. P5 explained that, by doing so, he could use his knowledge of, *e.g.*, button event listeners for voice input.
Design implication: instead of considering voice as a different mechanism, Geno should provide a unified process for developers to realize an interactive task across any combination of voice and pointing modalities.

(iii) None of the participants attempted or were able to develop an error handling mechanism—that is, when an intended voice input is not recognized by an API, what then? Foremost, participants were often confused about why certain utterance was not recognized as the intended interactive behavior. By default there is no feedback from the API whether the app is listening at all, or if so, what is heard and how does that (not) trigger the intended action of the application.
Design implication: Three levels of feedback are useful for developing NLP-enabled interaction: whether the app is listening, what is heard and what is (not) being acted upon; further, there should be support for developers to handle misrecognition errors, *e.g.*, using a dialog to ask for disambiguation or extra information from the user when ambiguity occurs.

(iv) Last but not least, we were surprised to find that three participants at the beginning were bogged down by the inability to access the microphone on the experiment laptop. P3, P4, and P5 experienced issues with the microphone and audio interface. P4 struggled with permission issues, and P5 had runtime errors due to an undetected microphone, which he later discovered was caused by incorrect declaration and misplacement of the voice recognition object.

Design implication: although not the core of multimodal input, low-level hardware access should be integrally supported as part of the Geno toolkit and should be abstracted away from developers' main workflow.

GENO: ADDING MULTIMODAL INPUT TO EXISTING APPS

In this section, we describe how Geno works: *(i)* how to create an intent at development time and *(ii)* what is the run-time behavior of an intent. Specific implementation details are discussed in the next section.

Development Time

With Geno, the main workflow of adding an intent to an existing web app is guided by a dialog (Figure 2) that involves the following steps: specifying target action, configuring voice input, and specifying GUI context.

Step #1: Specifying a Target Action

Geno supports adding a voice modality to two types of target actions—an intent can invoke either a JavaScript function, or a sequence of GUI interactions on the interface of the web app. Sometimes these two types are equivalent (*e.g.*, calling a `changeColor` function *vs.*using a color picker widget), but in general, GUI interactions allow for one-off, nonparametric input (*i.e.*, simply using voice to invoke the same input sequence, such as clicking on the "next song" button) whereas functions' behavior can be varied based on specific contents in the voice input (*e.g.*, moving a calendar event to "next week", "Friday" or "tomorrow").

To specify a target action that executes an existing function, a developer clicks the ● next to a function (Figure 3a), which creates a new intent and opens a dialog for configuring the voice input with the function/arguments pre-filled as the target action/slots (details in Step #2 below).

It is possible that a single function for an intended target action does not yet exist. For example, as shown in Figure 1, moving an event to a new date in the calendar app requires invoking multiple existing functions to retrieve an `Event` object from the title of the event, create a `Date` object from the user's natural language description of the target date, and set the `date` field of the `Event` object using the `setDate()` function[9]. In this case, Geno allows a developer to create an intent without a specific function by clicking the ✚ button (Figure 3b) to initiate the same dialog, go through the following steps (#2 and #3), and then Geno automatically generates a skeleton of the new function with the function name and its list of arguments so that the developer can implement the body of the function. In a function, the developer can also use Geno to say the result of the action by speech using the `geno.say()` function.

Currently, Geno only supports associating voice input with one function at a time; in the future, we plan to explore multiple functions and their different relationships with respect to a voice input (detailed in discussion).

To specify a sequence of GUI interactions as a target action, a developer would instead focus on the web app preview

[9]Note that programming such logic is application-specific, thus not automated by Geno.

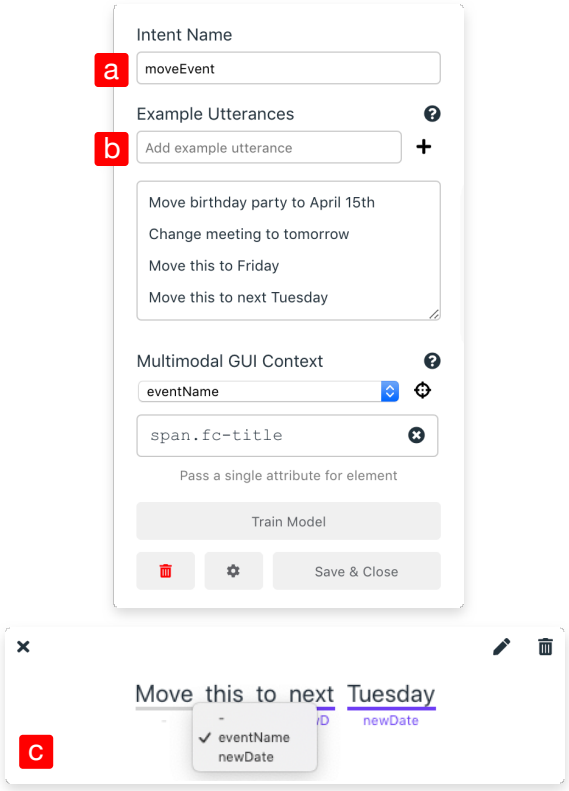

Figure 2. An overview of Geno's workflow, which is guided by this dialog: after specifying a target action, a developer creates an intent (a), provides example utterances (b), labels extracted entities as parameter values (c), and optionally adds GUI context to supplement the voice modality.

and click on the ➤ (Figure 4a) button to demonstrate a sequence of actions on the GUI, such as clicking the 'week' radio button to switch the calendar's view (Figure 4b), or to open a color picker and select the color red. After the developer enters the demonstration mode, Geno displays a blue highlight overlay above the element currently hovered over by the mouse cursor and a description of this element to help the developer identify the correct target element (Figure 4b).

Step #2: Configuring Voice Input

The next step in creating an intent to configure its voice input. This includes *(i)* providing sample natural language utterances for invoking this intent; and *(ii)* labeling the task parameter values in the sample utterances.

To add new sample utterances (Figure 2b), a developer types in two to three example utterances, *i.e.*, different ways of saying a command to invoke this intent, such as "reschedule this to next week", "move *Birthday Party* to 6PM today", "shift *Group Meeting* to Friday". Here we chose typing in rather than speaking out sample utterances, as typing makes editing easier while composing an utterance. These sample utterances are used for training an intent classification model that, at run-time, can associate the user's voice commands with the corresponding intents (details in the Implementation section).

After adding the sample utterances, the developer labels the parameter values in those utterances. As shown in Figure 2c,

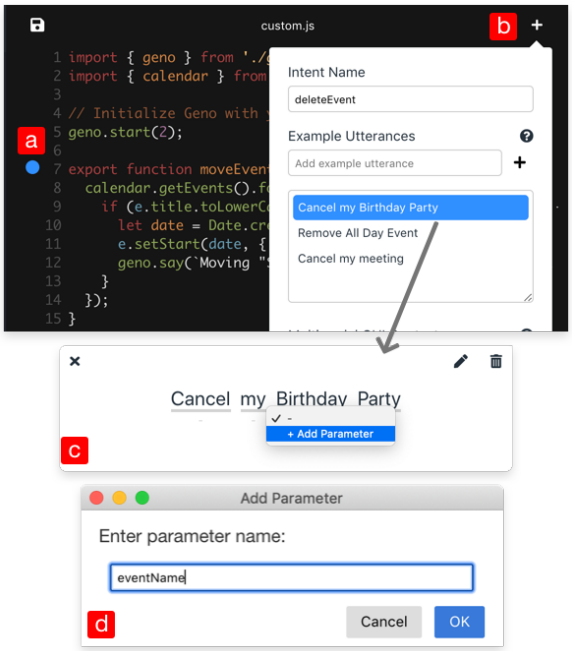

Figure 3. #1 Specifying a target action as a function, which can be one that already exists (clicking the ● aside a function) (a), or a new one (clicking the ✚ button) (b). For a new function, Geno allows a developer to create a new parameter, using entities in an utterance as the arguments of the new function.

for each utterance, the developer can click on a part of the utterance (*e.g.*, "Tuesday") and label it as the value for a parameter (e.g., "`newDate`"). If the intent was created from an existing JavaScript function, then the list of parameters will be automatically populated with the list of arguments in the function (Figure 1a). Otherwise, if the target function has yet to be implemented, the developer needs to define new parameters as they label the parameter values (Figure 3cd). Those labels are used for training an entity extraction model for extracting task parameter values from the user's run-time voice commands (details in the Implementation section). Note that a target action can be nonparametric (*e.g.*, when the target JavaScript function does not have any arguments or when the target action is a sequence of GUI interactions), in which case the developer can skip labeling parameter values.

Step #3: Adding GUI Context

Geno supports multimodal inputs that use both voice inputs in natural language and mouse inputs of referencing to GUI context. For example, a user might say "reschedule this to next week" while pointing to an event on the calendar with the cursor. In this case, the reference of "this" specifies the value of the parameter `eventName` in this intent (`moveEvent`). Currently, Geno supports extracting such information from GUI contexts either as a single HTML element hovered over by the cursor or multiple elements marquee-selected by dragging. Multi-selected elements are extracted as lists, which is useful for supporting commands such as "adding *all of these* to the playlist" while selecting multiple songs.

To add the support for GUI context input, the developer first selects a target parameter (e.g., `eventName`) from the "Multi-modal GUI Context" dropdown menu (Figure 5a), and then clicks on the ✥ icon (Figure 5b) to demonstrate an example GUI context. As shown in Figure 5c, in the calendar example, the developer first hovers the cursor over a calendar event, which highlights the HTML element as a potential selection, and clicking[10] an element would confirm it as GUI context. Once an element is selected, Geno extracts a list of its HTML attributes such as its class name and innerText (Figure 5d). The developer then selects one target attribute that contains the desired value. In the calendar example, the developer chooses the `innerText` field, since it contains the name of the event. This demonstration allows Geno to create a data description query [19] for the GUI context input, so that at run-time, when the user points to a GUI element or selects a list of GUI elements, Geno can extract the correct attributes from these elements and use them as parameter values in the detected task intent.

Run-time

At run-time, Geno floats over the application (Figure 1d) on the user's end. A user can click the 🎤 icon or use a built-in keyboard shortcut (we use `Ctrl + ` `) to start speaking a voice command, at times in tandem with references to GUI context. Geno transcribes the user's speech to text and matches the text to an existing intent created by the developer. If Geno cannot find a matching intent, it will say "Sorry, I didn't understand. Could you try again?" to prompt the user to provide a new utterance.

In order to execute the target action of the matching input (*e.g.*, `moveEvent`), Geno will need to find the requisite parameters (*e.g.*, `eventName` and `newDate`). For each parameter:

(i) If an entity corresponding to the parameter is found in the utterance, Geno will "fill" the target action with the entity;

(ii) If no entity is found, Geno first checks whether the GUI context has been configured for that parameter, and if the user has hovered over a matching GUI context with the voice command. If so, Geno extracts the parameter values from the user-specified GUI context (*e.g.*, the `innerText` of an HTML element for an event).

(iii) If no GUI context is found, Geno will ask a follow-up question for each missing parameter, which the user can directly respond to, *e.g.*, "What is eventName?", although the developer can customize the prompt question for each parameter in the options view in the popover (accessed by clicking ⚙ button) where each parameter is listed with a text field to specify a custom question.

Once all the required parameters have been "filled", Geno executes the target action by either invoking the corresponding JavaScript function or replaying the sequence of demonstrated GUI interactions as configured by the developer.

IMPLEMENTATION

In this section, we describe key implementation details that underpin Geno's workflow of creating multimodal input on

[10]Native clicking events on the web app were temporarily disabled while specifying GUI context.

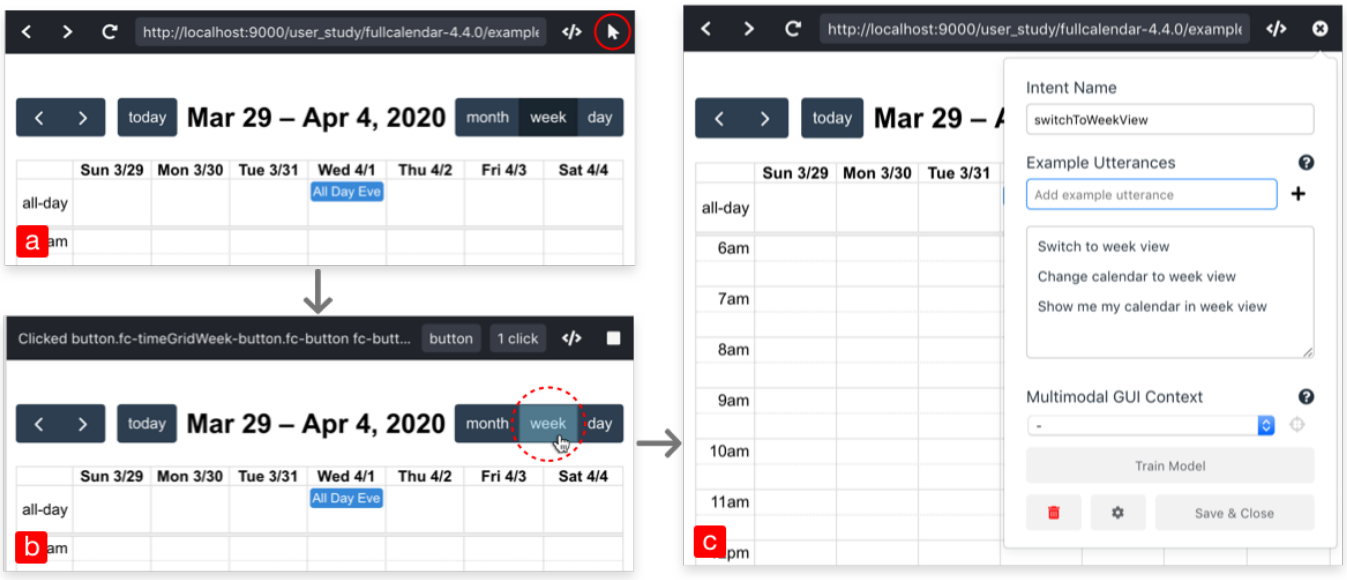

Figure 4. #1 Specifying a target action (by demonstrating GUI interactions): clicking the button starts the recording of a sequence of input events and recognizes clicking the 'week' button as a click action (b), based on which an intent is created to trigger the replay of the recorded click action (c).

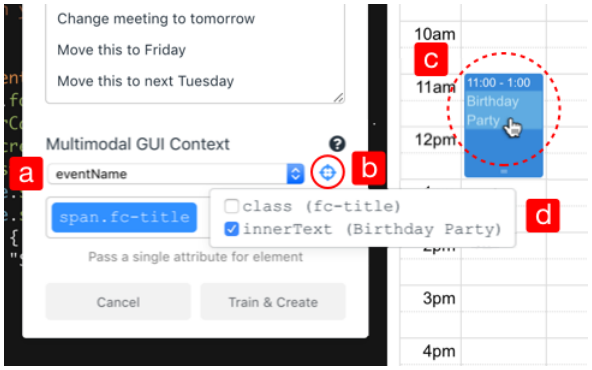

Figure 5. #3 Adding GUI context: first select a parameter `eventName` (a), then start a demonstration (b) and click or drag to select a calendar event element (c) and its attribute `innerText` which can be used as a value for `eventName`.

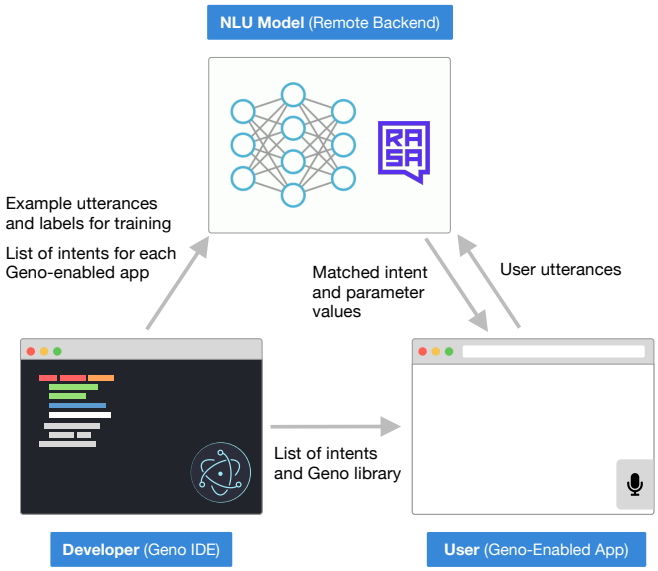

Figure 6. An overview of Geno's system architecture.

existing web apps. Figure 6 shows an overview of Geno's system architecture.

Natural Language Processing Pipeline

Geno uses a typical frame-based dialog system architecture for processing the user's natural language commands. After the user speaks an utterance, Geno transcribes the speech to text using the Web Speech API's[11] SpeechRecognition toolkit. A natural language understanding (NLU) module then classifies the utterance into one of the intents defined by the developer and extracts the parameter values from the utterance.

The intent classifier uses the StarSpace model [37], which is a state-of-art general-purpose neural embedding model for text classification. Given a voice query, the classifier returns a ranking of potential intent matches based on confidence of the match. We look at the match with the highest confidence, and execute the command if it meets a 50% minimum threshold of confidence. The parameter value extractor uses a CRF entity extraction model[12] to recognize named entities (*e.g.*, date, location) in the user utterance. Both models run on a remote server and communicate with the client-side of Geno and Geno-enabled web apps through HTTP requests. The models are implemented using the open-sourced Rasa library[13].

The intent classification and the parameter value extraction models are trained using the example utterances and their la-

[11] https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API/

[12] https://rasa.com/docs/rasa/nlu/entity-extraction/

[13] https://rasa.com/

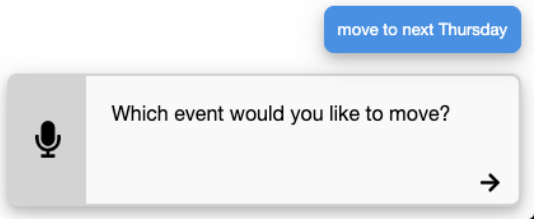

Figure 7. If Geno cannot find a parameter value in any modality, it falls back to prompting the user to specify the parameter value verbally.

beled parameters provided by the developer (Step #2). Specifically, the back-end first creates tokens from input sentences, encodes input sentences into vector representations, and then creates bag-of-word representations for model training.

For intents with parameters that support GUI context input, Geno's parameter value extractor looks for common demonstrative pronouns (*e.g.*, this, that) in the user's utterance. If found, the system will look for relevant HTML elements, extract attributes predefined by the developer (Step #3) and fill in the corresponding parameter with the value from the GUI context. For example, if the user says "Move *this* to Friday" while pointing at the event with the title "Birthday Party" for the Geno intent shown in Figure 2, Geno would detect the reference of "this" in the utterance, finds a calendar event element based on the hovering mouse position and retrieves its value "Birthday Party" from the event element's `innerText`, and uses it for the value of the task parameter *eventName*.

For each unfilled parameter in the intent, Geno asks a follow-up question until all the parameters for the intent are filled, so the intent is ready for execution. Geno uses Web Speech API's SpeechSynthesis library to ask these questions both by voice and by displaying the text in the slide-over window (Figure 7)

Recording and Replaying GUI Demonstrations

When the developer demonstrates GUI interactions as a target action (Step #1), Geno records mouse clicks and keyboard input. Although it is possible to click on any HTML element, not all of them represent meaningful actions intended by the developer. Thus Geno only records interaction with clickable elements by filtering out elements that do not have a `.click()` function. The recorded actions are saved as a list where each entry is a clicked element's tag and index, which can be used to retrieve the same element at run-time (*i.e.*, getting the `[index]`th element of type `[tag]` in the DOM tree) and programmatically invoke a `click` event on the element or enter text into the element.

Extracting GUI Context

The developer creates the support for using GUI context as a parameter by demonstrating in the web preview (Step #3). Geno identifies HTML elements as GUI context by detecting hovering over or dragging to marquee-select elements. Specifically, hovering is detected by when the mouse cursor displacements between consecutive `mousemove` frames are smaller than a threshold and the hovered element is retrieved using `document.elementFromPoint(x,y)`; dragging is detected when a displacement between `mousedown` and `mouseup` is greater than a threshold and elements completely within the rectangular dragged area will all be considered as GUI context.

Once the element(s) are located, Geno extracts the identifiable features for each selected element by its `tag`, `class`, and `element.attributes`. The developer can select one or more such features to create an attribute filter for this GUI context parameter (Step #3, Figure 5). This attribute filter is used for extracting the parameter value from the user's selected GUI elements at run-time. For example, when the developer clicks on the "Birthday Party" element in the calendar app to demonstrate the GUI context input for the parameter `eventName`, as shown in Figure 5, Geno extracts `fc-title` (the `class` of this element) and "Birthday Party" (the `innerText` of this element). The developer chooses the `innerText` attribute to indicate using its value for the `eventName` parameter. At run-time, when the user hovers over a HTML element while saying a command that triggers the `moveEvent` intent, Geno finds the hovered HTML element that matches the selector `span.fc-title`, extracts the value of its `innerText` attribute, and uses this value for the parameter `eventName` when invoking the target JavaScript function for the intent `moveEvent`.

Integrating with Existing Web Apps

When a developer loads an existing web app into Geno, the system creates a directory in the root of the project folder that includes a `geno.js` file that supports all the requisite functionalities in Geno-enabled apps and a JSON file that contains all the developer-specified intents.

After the developer specifies the intents and trains the NLP models, Geno automatically builds the project, updates the JSON file and imports `geno.js` and other requisite files into the web app. As shown in Figure 6, when a user runs a Geno-enabled app in the browser, `geno.js` gets executed, which adds the Geno voice command button to the interface, manages the intents stored in the JSON file, communicates with the back-end server to understand the user's voice commands, and facilitates the execution of the target actions for user intents. Geno uses dynamic imports in JavaScript to execute functions with associated voice commands.

Environment & Software Toolkits

Geno uses Electron[14] as the core for its IDE. Electron allows desktop apps to be built in JS and also supports cross-platform compatibility for macOS, Windows and Linux. Geno was written in TypeScript and React[15] and uses Acorn[16] for parsing AST of JavaScript files to identify functions, various React components for UI elements (*e.g.*, CodeMirror[17] for code editor, Treebeard[18] for file explorer), Lowdb[19] for database related tasks, and Chokidar[20] for watching files on disk and

[14] https://www.electronjs.org
[15] https://reactjs.org
[16] https://github.com/acornjs/acorn
[17] https://codemirror.net
[18] https://github.com/storybookjs/react-treebeard
[19] https://github.com/typicode/lowdb
[20] https://github.com/paulmillr/chokidar

keeping UI components up to date for any database-specific state changes.

EXISTING WEB APPS MULTIMODALIZED BY GENO

To demonstrate the expressiveness and practicality of Geno, we created multimodal input on five examples of existing actual websites and web apps, as shown in Figure 8. The New York Times: Search for related news articles on a website by hovering over interesting text; FoodNetwork: Hands-free control of a recipe website while cooking, *e.g.*, play/pause or skip through a walkthrough instructional video and have Geno read out the steps; Three.js: Switch between modes and manipulate 3D models by voice commands in 3D modelling software; Expedia: Quickly search for flights by referring to airport codes; Yahoo! Mail: Manage email more efficiently using multimodal input by dragging over emails and saying to "Delete these", or forwarding emails by hovering over an email and saying "Forward this to Alex".

DEVELOPER STUDY

We conducted a developer study with the research question to validate the usability and usefulness of Geno for developers to create voice+GUI multimodal input on existing web apps.

Participants

We recruited eight participants aged 19-26 (mean 22.6, SD 2.1). All participants were male, five were students, and three were professional software engineers. We asked the participants to self-report their web development experience: four participants considered themselves expert in web development, one was intermediate and three were novice. Only one participant had prior experience developing voice interfaces. Each participant received a $20 Amazon gift card for their time.

Apparatus

We provided participants the same starting codebases as the ones used in the formative study. Due to the COVID-19 pandemic, all study sessions were conducted remotely online via Zoom[21]. Specifically, we set up Geno in one experimenter's laptop and used Zoom to allow each participant to remotely control the laptop and interact with Geno to perform the development tasks. We screen-recorded (including audio) the study sessions also using Zoom.

Procedure & Tasks

Each one-hour study session consisted of a brief introduction to Geno, a walkthrough tutorial, two development tasks, a questionnaire, and a semi-structured interview. We used a simple text editor app to introduce the goal of Geno and walked through an educational 'Hello World' example where we showed each participant how to create voice input to change the color of text and to toggle the 'bold' formatting button.

The main development tasks mirrored our formative study except that for each task, we added an additional input that requires *both* voice and GUI inputs, and each participant was asked to complete tasks in *both* applications (as opposed to just one in the formative study).

- Task 1–Calendar: create input to
  (i) use voice to change to week view
  (ii) mouse-hover over an event and use a voice command to reschedule the event to *N* days later;

- Task 2–Music player: create input to
  (i) use voice to skip a track
  (ii) mouse-hover over a song and use a voice command to add the song to a playlist;

For each input in the assigned task, we provided the participant with five different test cases. For example, a test case for the second calendar input would be hovering the mouse over a "Meeting" event on Tuesday 10 AM this week and saying "Postpone this by three days". After a participant finished developing each input, one experimenter tested the result by walking through and performing the pre-defined test cases. We did not let the participants directly speak to Geno due to the often poor audio quality in the remote study, which affected the speech recognition performance. Instead, we asked the participants to test the app by telling the experimenter what to say, and the experimenter relayed the utterance for them.

At the end of the tasks, each participant was asked to answer a questionnaire regarding the easiness and usefulness of using Geno. Based on their responses, we conducted a short semi-structured interview to understand where they felt the development was well-supported, what remained challenges and how they would see Geno to be improved in the future.

Results & Findings

There were 2 multimodal interactions $\times$ 2 applications = 4 development tasks for each participant. Across our study, six users completed all four tasks in less than an hour, one user completed three tasks, and one completed two. The two developers who did not complete all the 4 tasks self-reported as novice web developers, and spent longer time writing and testing the JavaScript functions during the study.

We conducted a thematic analysis of the qualitative data [5]. Overall, Participants appreciated the mission of Geno: "*I would definitely say it's very useful ... addressing the accessibility issue for most of the people who need it.*" (P7)

Participants also appreciated how Geno simplified the programming of multimodal input with a GUI-based guided workflow, achieving the "*right balance of abstraction*" (P2):

> *... provides like a GUI on how to adding interactions to your application.* (P5)

> *It does so much for you. I just kind of forget that I actually have to do anything.* (P2)

Participants found Geno overall easy-to-learn.

> *I just saw one example, and I was able to do it.* (P8)

> *it was I think initially was a little hard, but I think I got it later ... the learning curve for this (Geno) is much better (than learning a library)* (P6)

[21] https://zoom.us

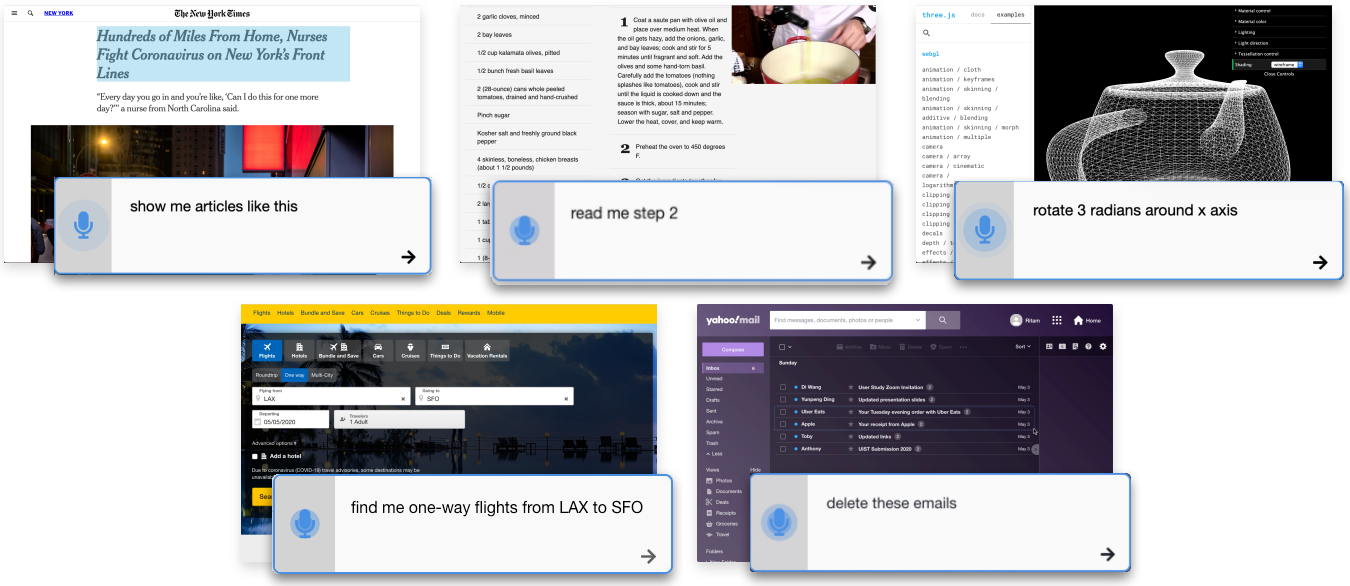

Figure 8. More voice+GUI input examples created using Geno based on actual websites.

Importantly, Geno was considered "*a very useful tool*" (P4) that "*definitely breaks down the barrier of traditionally you think adding anything machine learning related to your product is a very hard task.*" (P5)

NLU support is the most appreciated aspect of Geno
Participants were overall impressed by how Geno made it much easier to program NLU into existing applications than what they had expected.

> *... adding a couple of examples and it knew exactly what you are talking about.* (P1)

> *I definitely would not want to do it myself so I would definitely be in favor of using something that would do most of it for me.* (P2)

Specifically, despite a lack of NLP knowledge, participants were able to understand and learn how to provide example utterances and labeling entities:

> *... weren't necessarily like intuitive, but I mean, once you learned it, it was pretty usable.* (P7)

> *... it was like a lot easier than I thought it was going to be. I thought I was gonna have to like do some complicated things, but I didn't really have to do anything that complicated.* (P2)

Below we summarize the main challenges that the participants faced when using Geno to create multimodal input on existing web apps.

The challenge of specifying target function: function vs. GUI demonstration—when to use which
We found participants sometimes confused about which approach to take. For example, when implementing multimodal input to move a calendar event, P1 first attempted to use demonstration, only to realize it was not possible to drag an event out of the current view. Participants also might not have realized that the support for demonstration currently is still limited to one-off action replay that cannot be parameterized. We believe such confusion can be prevented with a tutorial with a set of comprehensive examples to explain the differences between specifying function and demonstrating GUI interactions, which we leave as future work. We will also explore merging these two approaches, *e.g.*, automatically extracting function calls from GUI demonstration so that a developer can associate voice input to a specific function and enable parameterization.

The challenge of GUI context: which elements to choose (dev-time) and which elements to hover (run-time)
Participants thought it was "*convenient*" (P1) that Geno could automatically detect a UI element as the pre-defined GUI context. However, some participants had difficulty deciding which element to use as the GUI context. For the music player app, P4 chose the wrong element that did not contain information about the song (to be added to the current playlist); during testing, P4 forgot about which element s/he was supposed to hover. For not choosing the right element as the GUI context, we believe it is part of trial-and-error process typical in learning to use a new development tool, which can be better supported in the future, *e.g.*, iteratively testing multiple elements and their attributes *in situ* as a part of Step #3. For not knowing which element to hover at run-time, future versions of Geno can provide visual cues to indicate elements that have been pre-defined to serve as the GUI context.

The challenge of GUI context: scalability
P7 pointed out that scalability could be another issue if the UI became more complicated, as hovering over an intended element would be challenging when there were multiple small surrounding or nesting elements. To address the scalability issue, one possible solution for future work is to enable coarse

selection of context: instead of having to hover over the exact element, Geno should allow for imprecise pointing and automatically search for the element that matches the type specified by the developer.

The challenge of striking a balance between abstraction and transparency
As Geno abstracts away details of multimodal input from non-expert users, it inevitably creates 'black boxes' in the process where questions from developers might arise. For example, when adding GUI context, P5 questioned the priority between processing voice and mouse inputs (we used a canonical frame-filling approach). Even for the NLU part, multiple questions were raised. P1 wanted to find out the "bound" of the NLU model: *"If I can understand the rule of NLU better, it can help..."* (P1); P8 questioned how he could know when to stop adding utterances. We believe in future work, Geno can provide more information to better inform non-expert users of Geno's process, *e.g.*, a dynamic visualization of the frame-filling approach, intermediating the performance of the NLU model as a developer provides more example utterances. Importantly, lest to overwhelm developers with too much detail, Geno should allow developers to request such explanatory information on demand.

LIMITATIONS, DISCUSSION & FUTURE WORK

We discuss the current limitations of Geno that point to opportunities for future work.

Specifying a Target Action

When specifying a target action as a JavaScript function, Geno currently only supports using one function for each intent, although a developer can bypass this limitation by writing a wrapper function that invokes multiple subroutines. In the future, we plan to explore the support for specifying multiple functions and their different relationships with respect to voice input. For example, an action can consist of a sequence of function calls or calling different functions based on specific conditions.

When specifying a target action as a demonstration of GUI interactions, Geno currently only supports recording and replaying nonparametric clicking events. In the future, we will support demonstrations that can be parameterized. For example, a demonstration for "Search for headphones" would consist of typing "headphones" into a search box and clicking the search button. In this case, the text entered can be considered a parameter that can be changed at run-time, *e.g.*, "Search for speakers" would replace the contents of the search box with "speakers". Instead of simply replaying the recorded events, it is also possible to parameterize the sequence, which is demonstrated in Chen and Li's Improv framework [6] and will be explored in our future work.

Geno's mechanism of recording GUI interactions only supports clicking and text entry. It does not support interacting with widgets like sliders, knobs, date pickers, and spinners. It also does not record gesture inputs (drag & drop, flicking etc.) The replaying mechanism relies on the tag and the order of HTML elements to locate the target element; therefore it may break for some dynamically-generated or adaptive web interfaces. In the future, we plan to extend Geno's support for various GUI widget types. We will also expand Geno's replaying mechanism to support using flexible queries to locate the target HTML element at run-time using approaches such as [19].

Adding GUI Contexts

Although Geno showcases the usefulness of GUI contexts, sometimes HTML elements' attributes cannot provide sufficient information for a specific parameter. For example, in the calendar app's case, the GUI context would not work if multiple events share the same title. Although Geno is able to read from the user's references to GUI context and use them for invoking target actions, it is up to the developer to decide whether and how it is possible to make use of such information.

Another limitation is that Geno currently does not check the validity of parameter values when extracting them from voice utterances or GUI contexts. which may encounter exceptions at run-time when the user, for example, says "move the event to [a non-date value]". In future work, we plan to provide further exception handling and debugging support. For example, exception handling code can be automatically generated. When an parameter value exception occurs, Geno will automatically fall back to manually ask the user for that parameter.

Further, it is also possible that a GUI element might be too small, making it difficult to specify as context. Future work can employ a increasingly broader range of searching to match users' GUI action to the specified contextual elements.

Supporting a Wider Range of Web Apps & Platforms

So far, we have only tested Geno on single-paged, desktop web apps , which is a reasonable starting platform for deploying Geno-generated multimodal interaction. Our future work will engineer the current implementation to support larger, more complex multi-page web apps, *e.g.*, allowing for carrying over GUI context from one page to a function call that takes place on another. To support mobile platforms, we will have to redesign how to obtain GUI context since there is no hovering (Input State 2) on touch screens. Dragging, on the other hand, is already supported on the mobile platform. In addition, future work should reach out to more expert developers beyond our novice participants

Model Generalizability

Each NLU model is limited to a single application, since the model's intents are associated with functions unique to the codebase. However, in the future, we could create models for categories of applications (*e.g.*, calendar, food orders) that could be repurposed by multiple developers. The GUI demonstration also requires additional information to generalize to different commands. For example, if a user has already defined a command for "move this meeting to next Tuesday" and would like to add a second question "move this meeting to next Tuesday and change duration to 1 hour", then a new rule would be required for the second question since it includes a new type of information, *i.e.*, event duration.

ACKNOWLEDGEMENT

This work was funded in part by the National Science Foundation under grant IIS-1850183. We thank our study participants and the reviewers for their valuable feedback.

REFERENCES

1. Masahiro Araki and Kenji Tachibana. 2009. Multimodal dialog description language for rapid system development. In *Proceedings of the 7th SIGdial Workshop on Discourse and Dialogue*. Association for Computational Linguistics, 109–116.
2. Richard A Bolt. 1980. *"Put-that-there": Voice and gesture at the graphics interface*. Vol. 14. ACM.
3. Marie-Luce Bourguet. 2002. A toolkit for creating and testing multimodal interface designs. *companion proceedings of UIST* 2 (2002), 29–30.
4. Marie-Luce Bourguet. 2003. Designing and Prototyping Multimodal Commands.. In *Interact*, Vol. 3. Citeseer, 717–720.
5. Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
6. Xiang 'Anthony' Chen and Yang Li. 2017. Improv: An Input Framework for Improvising Cross-Device Interaction by Demonstration. *ACM Transactions on Computer-Human Interaction (TOCHI)* 24, 2 (2017), 15.
7. PR Cohen, M Darlymple, FCN Pereira, JW Sullivan, RA Gargan Jr, JL Schlossberg, and SW Tyler. Synergic use of direct manipulation and natural language. In *Proc. Conf. human Factors in Computing Systems (CHI'89)*. 227–233.
8. Philip R Cohen, Michael Johnston, David McGee, Sharon Oviatt, Jay Pittman, Ira Smith, Liang Chen, and Josh Clow. 1997. Quickset: Multimodal interaction for distributed applications. In *Proceedings of the fifth ACM international conference on Multimedia*. ACM, 31–40.
9. Bruno Dumas, Denis Lalanne, and Rolf Ingold. 2009. HephaisTK: a toolkit for rapid prototyping of multimodal interfaces. In *Proceedings of the 2009 international conference on Multimodal interfaces*. 231–232.
10. Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G Karahalios. 2015. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 489–500.
11. T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (June 1996), 131–174. DOI: http://dx.doi.org/10.1006/jvlc.1996.0009
12. Michael Johnston, Paolo Baggia, Daniel C Burnett, Jerry Carter, Deborah A Dahl, G McCobb, and D Raggett. 2009. Emma: Extensible multimodal annotation markup language. *World Wide Web Consortium, W3C Recommendation* (2009).
13. Michael Johnston, Giuseppe Di Fabbrizio, and Simon Urbanek. 2011. mTalk-A multimodal browser for mobile services. In *Twelfth Annual Conference of the International Speech Communication Association*.
14. Runchang Kang, Anhong Guo, Gierad Laput, Yang Li, and Xiang 'Anthony' Chen. 2019. Minuet: Multimodal Interaction with an Internet of Things. In *Symposium on Spatial User Interaction, SUI 2019, New Orleans, LA, USA, October 19-20, 2019*. 2:1–2:10. DOI: http://dx.doi.org/10.1145/3357251.3357581
15. Tejaswi Kasturi, Haojian Jin, Aasish Pappu, Sungjin Lee, Beverley Harrison, Ramana Murthy, and Amanda Stent. 2015. The Cohort and Speechify Libraries for Rapid Construction of Speech Enabled Applications for Android. In *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. 441–443.
16. Kouichi Katsurada, Yusaku Nakamura, Hirobumi Yamada, and Tsuneo Nitta. 2003. XISL: a language for describing multimodal interaction scenarios. In *Proceedings of the 5th international conference on Multimodal interfaces*. 281–284.
17. Gierad P Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. 2013. Pixeltone: A multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2185–2194.
18. 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*. 6038–6049.
19. Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. APPINITE: A Multimodal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.
20. Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. PUMICE: A Multimodal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 577–589.
21. Toby Jia-Jun Li and Oriana Riva. 2018. KITE: Building conversational bots from mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 96–109.
22. Marilyn Rose McGee-Lennon, Andrew Ramsay, David McGookin, and Philip Gray. 2009. User evaluation of

OIDE: a rapid prototyping platform for multimodal interaction. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. 237–242.

23. Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.
24. Brad A. Myers, Amy J. Ko, Chris Scaffidi, Stephen Oney, YoungSeok Yoon, Kerry Chang, Mary Beth Kery, and Toby Jia-Jun Li. 2017. Making End User Development More Natural. In *New Perspectives in End-User Development*. Springer, Cham, 1–22. DOI: http://dx.doi.org/10.1007/978-3-319-60291-2_1
25. Zeljko Obrenovic and Dusan Starcevic. 2004. Modeling multimodal human-computer interaction. *Computer* 37, 9 (2004), 65–72.
26. Sharon Oviatt. 1999a. Mutual disambiguation of recognition errors in a multimodel architecture. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 576–583.
27. Sharon Oviatt. 1999b. Ten myths of multimodal interaction. *Commun. ACM* 42, 11 (1999), 74–81.
28. Sharon Oviatt, Phil Cohen, Lizhong Wu, Lisbeth Duncan, Bernhard Suhm, Josh Bers, Thomas Holzman, Terry Winograd, James Landay, Jim Larson, and others. 2000. Designing the user interface for multimodal speech and pen-based gesture applications: state-of-the-art systems and future research directions. *Human-computer interaction* 15, 4 (2000), 263–322.
29. Fabio Paterno, Carmen Santoro, Jani Mantyjarvi, Giulio Mori, and Sandro Sansone. 2008. Authoring pervasive multimodal user interfaces. *International Journal of Web Engineering and Technology* 4, 2 (2008), 235–261.
30. Bastian Pfleging, Michael Kienast, Albrecht Schmidt, Tanja Döring, and others. 2011. Speet: A multimodal interaction style combining speech and touch interaction in automotive environments. In *Adjunct Proceedings of the 3rd International Conference on Automotive User Interfaces and Interactive Vehicular Applications, AutomotiveUI*, Vol. 11.
31. Chris Schmandt, Mark S. Ackerman, and Debby Hindus. 1990. Augmenting a window system with speech input. *Computer* 23, 8 (1990), 50–56.
32. Marcos Serrano, Laurence Nigay, Jean-Yves L Lawson, Andrew Ramsay, Roderick Murray-Smith, and Sebastian Denef. 2008. The openinterface framework: A tool for multimodal interaction. In *CHI'08 Extended abstracts on human factors in computing systems*. 3501–3506.
33. Vidya Setlur, Sarah E Battersby, Melanie Tory, Rich Gossweiler, and Angel X Chang. 2016. Eviza: A natural language interface for visual analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 365–377.
34. Adrian Stanciulescu, Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, and Francisco Montero. 2005. A transformational approach for multimodal web user interfaces based on UsiXML. In *Proceedings of the 7th international conference on Multimodal interfaces*. 259–266.
35. Amanda Swearngin, Amy J Ko, and James Fogarty. 2017. Genie: Input Retargeting on the Web through Command Reverse Engineering. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 4703–4714.
36. Edward Tse, Saul Greenberg, Chia Shen, and Clifton Forlines. 2007. Multimodal multiplayer tabletop gaming. *Computers in Entertainment (CIE)* 5, 2 (2007), 12.
37. Ledell Yu Wu, Adam Fisch, Sumit Chopra, Keith Adams, Antoine Bordes, and Jason Weston. 2018. Starspace: Embed all the things!. In *Thirty-Second AAAI Conference on Artificial Intelligence*.