Genie: Input Retargeting on the Web through Command Reverse Engineering

Amanda Swearngin[1], Amy J. Ko[2], James Fogarty[1]
[1]Computer Science & Engineering, [2]The Information School
DUB Group, University of Washington
amaswea@cs.washington.edu, ajko@uw.edu, jfogarty@cs.washington.edu

ABSTRACT
Most web applications are designed as one-size-fits-all, despite considerable variation in people's expertise, physical abilities, and other factors that impact interaction. For example, some web applications require the use of a mouse, precluding use by many people with severe motor disabilities. Other applications require laborious manual input that a skilled developer could automate if the application were scriptable. This paper presents Genie, a system that automatically reverse engineers an abstract model of the underlying commands in a web application, then enables interaction with that functionality through alternative interfaces and other input modalities (e.g., speech, keyboard, or command line input). Genie comprises an abstract model of command properties, behaviors, and dependencies as well as algorithms that reverse engineer this model from an existing web application through static and dynamic program analysis. We evaluate Genie by developing several interfaces that automatically add support for speech, keyboard, and command line input to arbitrary web applications.

Author Keywords
Reverse engineering; web application; program analysis.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI).

INTRODUCTION
Many people rely on the web for diverse information needs, such as tracking news and events, communicating with friends, playing games, and monitoring personal finances. However, these same people have a diverse set of computer skills and physical abilities, often requiring them to interact with websites in ways designers did not anticipate. For example, for blind and low vision people, reading a web page requires using a screen reader [4]. Alternatively, people with severe motor impairments may be able to see a web page, but their limited ability to use a mouse may leave many interactive parts of the web almost impossible to use [13]. Other factors such as culture [28] and gender [30] can impact how a person perceives and interacts with a website. Websites are also viewed through a multitude of devices, resolutions, and form factors. It is infeasible for a developer to create a website that suits every need and every type of device.

What interfaces people perceive as usable often depends on their cultural background [28]. Reinecke et al. created culturally adaptive interfaces that corresponded to a 22% performance increase when compared to the original interfaces. A person's gender [20] can influence the appeal and trustworthiness of a website. A person's age can also affect perception, hearing, cognitive, and motor abilities, requiring new interface designs [17]. However, few of these factors are considered by developers when designing web applications.

Another area where diverse needs are not always addressed is in accessibility. Per a 2015 survey [31], 61.3% of screen reader users stated that web content has become *less* accessible or has not improved in the past year. The Web Content Accessibility Guidelines [8] provide a set of guidelines for making web content accessible, including ARIA attributes [10], which enable screen readers and other devices to translate and interpret interactive web content. Unfortunately, one study [27] found that only 50.4% of problems encountered by blind users were covered by the success criteria in these guidelines, and while 16.7% of websites implemented these guidelines, they did not solve accessibility problems. Moreover, studies of a large set of government and high-traffic websites found that most did not even implement the guidelines that *are* helpful [16].

Prior work has explored novel ways of extracting abstract models of user interfaces to make them more customizable to alternative needs. For example, Prefab [11] is a system that enables adding advanced behaviors on top of an existing interface, such as a target-aware implementation of the *Bubble Cursor* [23]. PAX [9] is a hybrid framework that enhances the capabilities of pixel-based systems, combining pixel-based information with the metadata exposed through the Accessibility APIs built into most operating systems. This approach enables more advanced interaction

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

ACM 978-1-4503-4655-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3025453.3025506

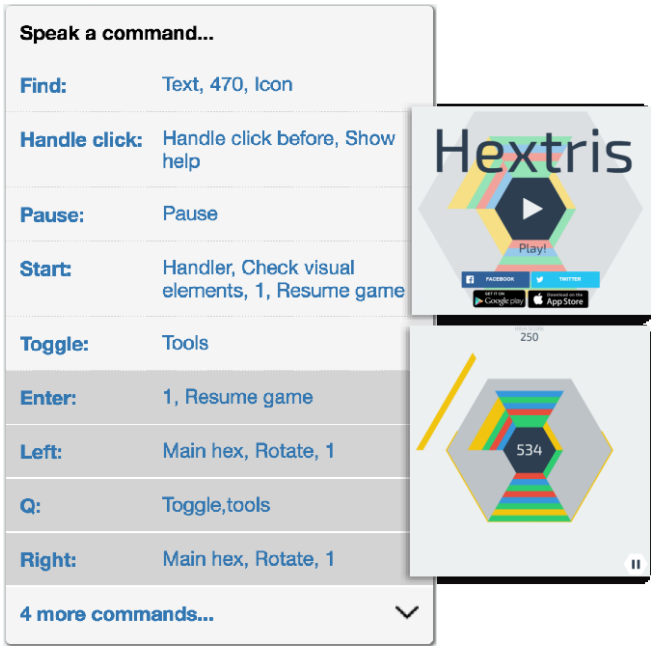

Figure 1. The Hextris web game (hextris.io) shown with a list of speech commands created by Genie. Speaking the bolded text label for each command triggers the corresponding actions.

techniques, such as "Screen Search" which enables searching for GUI components in an interface. SUPPLE [14] discovered the needs of individual users through a performance test and automatically generated a customized interface based on their preferences.

Although prior work demonstrates the power of generating abstract models of interfaces, building these models from surface features such as pixels and accessibility APIs is limiting. Interfaces can contain a variety of behaviors that cannot be discovered from visible features, including keyboard input, touchscreen gestures, and other custom commands. This is particularly true on the web, where websites use a vast array of different user interface toolkits and custom interactive controls that can vary by platform and device.

This work presents Genie, a new approach to interface modeling that applies program analysis techniques from software engineering to reverse engineer a model of an application's commands from source code. We explore this approach on the web, where reverse engineering is simplified due to the open access to user interface structure in the DOM and the source code that handles inputs. Genie derives models of the currently available commands from a web application and enables access to them through other input modalities. For example, Figure 1 shows a game that originally supported only mouse and keyboard input. Genie recovers the application's underlying command structure and provides an application-agnostic speech interface to access the game's commands. Speaking the command label *"Left"* (a command label derived by Genie's analysis) rotates the hexagon left, performing the same functionality as clicking the left mouse button or typing the left arrow key. This *input retargeting* approach allows a web application designed to support one input modality to be mapped onto any other input modality, ensuring that users can access any website according to their diverse needs.

Command Property Definitions

Available: Dependent on the *enabled* and *visible* state.

Enabled: True if at least one *data dependency* with at least one *side effect* is currently satisfied.

Visible: The command element is visible on the screen.

Data Dependency: A condition in an event listener that can be evaluated to have the value of either *true* or *false* and is associated with at least one *side effect*.

Side Effect: A statement in an event listener that has an effect on the system when executed.

Device Dependencies: A list of commands that must be performed before (*pre*) or after (*post*) a command based on the implicit device requirements.

Required Input: A list of required inputs to the command. Possibilities are *mouse location*, *mouse button*, *key code*, and *value*. Knowing each required input value allows Genie to generate them automatically, or request them from users, when required.

Perform*:* Triggers the command automatically, supplying the correct *required input* values, and triggering commands corresponding to the *pre* and *post device dependencies*.

Figure 2. Genie abstract data model property definitions.

The contributions of our work are:

- An application-agnostic abstract model of interactive commands and their properties.
- A method for reverse engineering these commands from an existing web interface into this abstract representation.
- An API that exposes a website's commands, which can be used to author interfaces that support other input modalities.
- Several application-agnostic interfaces that automatically retarget input to add speech, keyboard, and command-line input capabilities to arbitrary web applications.

In the rest of this paper, we describe the architecture and implementation of our approach and then demonstrate the potential of Genie in several examples of input retargeting. We end with a discussion of how Genie builds upon prior work, additional remaining challenges of input retargeting, and several ideas for how Genie can augment a website to support diverse needs.

THE GENIE FRAMEWORK

Genie models web application user interfaces as a set of available *commands*. Each command has a set of *properties* (shown in Figure 2), which represent command availability, dependencies, and other metadata. Genie periodically refreshes commands and their properties as the state of the interface changes in response to user interaction or other application background activity. Genie also provides generic support for

executing each command, allowing the implementation of alternative interfaces that support other input modalities.

Genie is implemented as a Google Chrome extension that accesses the web page DOM, event registrations, and source code. Genie consists of five algorithms to 1) *detect* commands, 2) *filter* commands to those that are directly executable, 3) *analyze* properties of commands to update their status in an interface, 4) *describe* commands obtain appropriate labels for an interface, and 5) *invoke* commands by recreating their inputs and event sequences. The following sections describe each algorithm and how they discover and update the properties and behaviors of the Genie data model.

Command Detection

The Genie system interposes application event registration to detect commands. Genie assumes graphical user interfaces consist of graphical *elements* that make up the interface, *events* that are triggered by input devices performing various actions on elements (e.g., `click`, `keydown`), and *listeners* that receive and respond to events. This event-subscription model is the dominant way of receiving and responding to input in modern user interface frameworks. Web interfaces also consist of a set of default *interactive elements* such as links (i.e., `<a>`) or input fields. Genie maps each *interactive element* and registered *event listener* to a unique command.

Genie detects commands by intercepting each event listener registration as it occurs. The default DOM APIs, as described by the W3C (www.w3.org), do not provide a method of accessing all currently subscribed event listeners in the DOM. We therefore monitor each registered event listener by overriding the default DOM API for `addEventListener`. This override notifies Genie that a new listener has been registered, and calls the original `addEventListener` method to register the listener with the browser. Event listeners can also be registered using a secondary library (e.g., jQuery or D3). These libraries wrap the `addEventListener` method, so intercepting `addEventListener` calls is sufficient to capture event listeners registered using these libraries. The override is achieved by injecting a script into the original page before DOM initialization so that all registered event listeners are intercepted. The script then intercepts all event listeners registered after DOM initialization to keep a currently updated list of events, each of which corresponds to a new *command*.

The default DOM APIs provide a second method of registering event listeners through global attributes. These listeners are registered in two ways. One is through an inline attribute on the DOM element in the format `onclick="listenerName()"` where the attribute name can be `onclick` or any of the supported event types[1]. Applications can also register global event listeners using the format `element.onclick=listenerName`. Genie locates and detects all registered global event listeners at document initialization and monitors any updates to them using the MutationObserver API[2], which notifies Genie of any updates to element attributes in the DOM. Genie also collects and monitors *interactive elements* (e.g., `<input>, <a>`) using the MutationObserver API.

Command Filtering

There are hundreds of events that can be triggered in a browser. A small subset of these events corresponds to interactive actions that a user can trigger by clicking, touching, or typing a key. Genie distinguishes between events that can be triggered by a user action, and those that happen as side effects of user actions or are triggered by the system.

Genie categorizes events into three groups: *direct, indirect,* and *system. Direct* events are those that can be triggered by human actions (e.g., a mouse click - `mousedown`, a key stroke - `keydown`). *Indirect* events happen as a side-effect of triggering *direct* events (e.g., an element gains focus - `focusin`, a field loses focus - `blur`). *System* events are events that are triggered without user intervention, but occur during interactive use of the system (e.g., a resource failed to load - `error`, a resource has finished loading - `load`). Genie collects each event listener registered to each *direct* event as a command, and filters out all event listeners registered to *system* events. Genie monitors *indirect* event listener registrations but does not expose them as commands, instead using them in the *invoke* module which we will describe later. The output of the command filtering process is: (1) a set of registered direct and indirect event listener and element combinations, and (2) a set of default interactive elements.

Command Property Analysis

User interfaces can change at any time in response to user interactions, events, and system status. To give an accurate picture of what a person can do at any given point in time, and to prevent needless interactions with disabled commands, we analyze and monitor the visibility and enabled state of each command, keeping the *visible* and *enabled* states in Figure 2 current with the original user interface.

To discover the value of these properties, for each intercepted event listener registration, we capture the event type, event listener source code, and the DOM element the listener is associated with. For the `visible` property, we query the DOM element for its visibility through a set of properties that can hide any element, such as setting the `display` attribute to `none` or other standard methods of hiding elements. Elements can also be off-screen or opaque, and we use existing DOM APIs to inspect these forms of visibility.

A command can be *disabled* (`enabled=false`) in two ways. First, a DOM element can set the `disabled` attribute. However, this attribute is not required, so it is possible that an element will appear interactive even if current conditions in its event listener prevent it from having effect. The second way a command can be *disabled* is therefore if the conditions in its event listener that result in *side effects* are not currently

[1] *www.w3.org/TR/html-markup/global-attributes.html*[1]

[2] *developer.mozilla.org/en-US/docs/Web/API/MutationObserver*

Input: AST for handleStartGame()

```
function handleStartGame(e) {
     var startBtn = $("#startBtn");
     if(!startBtn.attr("disabled")){
          startBtn.attr("disabled",false);
          // Start the game
          // If there are lives remaining
          if(livesRemaining > 0) {
             startGame();
          }
     }
}
```

Output: data dependencies (DD) & side effects (SE)

DD: `!$("startBtn").attr("disabled") && livesRemaining > 0`
SE: `startBtn.attr("disabled",false), startGame()`

DD: `!$("startBtn").attr("disabled") && !(livesRemaining > 0)`
SE: `startBtn.attr("disabled",false)`

DD: `$("startBtn").attr("disabled")`
SE: `None`

Figure 3. An event listener that starts a game if the start button is not disabled and if there are any remaining lives. Genie discovers three data dependencies in its analysis of this listener.

satisfied. We define each condition as a *data dependency* and each method call, state update, event, or other response in an event listener as a *side effect.* Each event listener has one or more *data dependencies* and each *data dependency* can have one or more *side effects.* If none of the *data dependencies* of a listener are currently satisfied, a command will have no effect, and we mark the command's `enabled` state as `false`.

Our system discovers *data dependencies* by analyzing the event listener source code as shown in Figure 3. First, our system parses the event listener to construct an abstract syntax tree (AST). Our system traverses this AST to construct an expression for each data dependency. We compute data dependency expressions by tagging each variable node in the AST with the node where it was either previously defined or declared. Then, we locate each conditional (e.g., `if`, `while`, `switch`). Within each conditional expression (i.e., the expression that must be true to reach the path specified inside conditional block), we locate each variable identifier and replace it first with where it was last assigned, and secondly, where it was last declared if we do not find any other assignments besides the variable's declaration. This process continues recursively until we have replaced all identifiers in a conditional expression with their corresponding assignments or declarations.

Figure 3 shows an event listener that handles clicking on a game's start button. There are three possible *data dependencies* corresponding to the three potential paths through the code, labeled by "DD" under output. These expressions depend on the current state of the Start button and the number of lives remaining. The first expression is constructed by combining the first `if` conditional with the second nested `if` conditional. The `startBtn` reference in the

Input: AST for rotateHexagonLeft()

```
// Rotate the hexagon left
function rotateHexagonLeft() {
     if (MainHex && gameState !== 0) {
          // Rotate hexagon left
          MainHex.rotate(1);
          // Update the position counter
          MainHex.hexagonPosition--;
     }
}
```

Output: Rotate the hexagon left, rotate hexagon left, rotate, update the position counter, hexagon position

Figure 4. Inputs and outputs for command description algorithm for the rotateHexagonLeft event listener.

first conditional is resolved to the value `$("#startBtn")` based on where it was last assigned in the listener, which occurs on the previous line.

Each data dependency is associated with one or more *side effects.* Identifying side effects is important for interpreting and displaying to a person what effects the command will have when it is executed. In Figure 3, `startGame()` is a side effect. Each line of code that would be executed if a data dependency expression is satisfied is a potential side effect. This method of side effect detection is potentially accurate if each method call, state update, or response is causing an update to the state of the system, affecting future interactions. However, this may not always be the case. To improve this method, we could potentially identify output affecting statements (e.g., as in [21]), and link each method call or state update to its last assigned location outside of the event listener, linking it to its origin in the website's source code. These could be searched for output affecting statements, or Ajax calls that cause data changes. However, this method would mostly be an approximation.

A command is disabled (`enabled=false`) if none of its data dependencies are currently satisfied, if none of its data dependencies have side effects, and if it has no side effects outside of conditional expressions. For example, Figure 1 shows a set of four disabled commands, *Enter, Left, Q,* and *Right.* This event listener for the *Left* command, shown in Figure 4, consists of only a single data dependency (`MainHex && gameState !== 0`), and a two side effects (`MainHex.rotate(1), MainHex.hexagonPosition--`). Before the game starts, the value of `gameState` is `0`. After a person starts the game, its value is `1`, resulting in this expression evaluating to `true`, and allowing the side effects to occur when the event listener is called.

Command Monitoring

After reverse engineering each command's data dependencies and side effects, Genie evaluates them to determine each command's availability. This process runs in an *update service* which calls the JavaScript method `eval` to evaluate each data dependency in a global scope. The results of `eval` are sent to each Genie interface through the update service. For example, the speech interface shown in Figure 1 updates

the status in the interface by giving the disabled commands a grey color. Our analysis currently only resolves data dependencies in the local event listener scope (excluding expressions defined outside this scope such as `MainHex`). This is because it is not possible to access the closure of registered event listeners to recreate the scoping of these variables. In future work, we will explore how we can evaluate these expressions more accurately within the constraints of JavaScript scoping.

Genie's update service runs every second, evaluating the enabled and visible states of each command as a person interacts with the system, providing them with timely feedback about command availability.

Describing Commands

Alternative interfaces that display available commands need some way to describe the commands so that people know what effect each command will have before they invoke it. For example, Genie derives and displays command labels in the speech interface shown in Figure 1, giving each command a name and basic description of its effects.

To derive these labels, Genie identifies labels from command metadata by searching for natural language phrases starting with an imperative verb followed by a noun. For example, the phrases in Figure 1 include "Resume game" or "Show help". If a command label cannot be found in the "*verb noun*" format, Genie searches each metadata string for verbs and nouns to use as labels, if they can be found.

Genie collects collect command metadata from two sources: *element* and *listener* metadata. *Element* metadata comes directly from the attributes of a command's DOM element. *Global* attributes are those common to all types of DOM elements, which include `title`, `id`, and `class`. These three attributes frequently and conventionally contain semantic metadata to describe the object or concept that a DOM element represents. `<input>` elements have an additional set of commonly useful attributes: `placeholder`, `alt`, and `value`. Another subset of elements, `<input>`, `<button>`, `<fieldset>`, `<textarea>`, and `<select>` have a `name` attribute. `<a>` elements have an `href` attribute that also frequently contains descriptive metadata.

Listener metadata comes from command event listener source code. Genie analyzes four sources of listener metadata: 1) comments on the event listener, 2) the event listener name, 3) comments on expression calls and assignments (i.e., side effects), and 4) the expression calls and assignments. To parse these sources, the algorithm splits each assignment, function call, and function name into separate identifiers (e.g., each token in Figure 4, such as `MainHex` and `hexagonPosition`), parsing each identifier and attribute value separately and identifying them as either a phrase or an individual word.

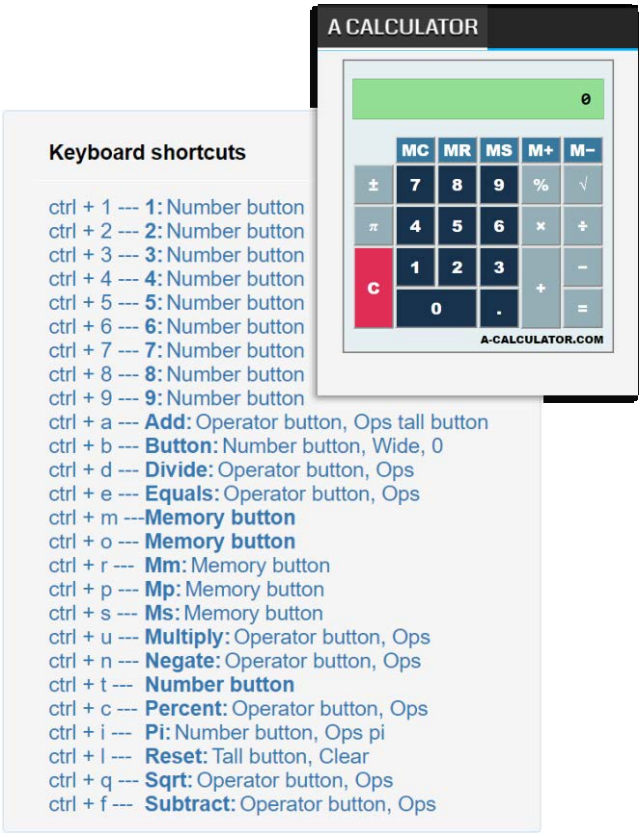

Figure 5. A calculator interface with incomplete keyboard support (a-calculator.com), enhanced to provide a keyboard shortcut for each command, as enabled by Genie's analyses.

Genie identifies phrases from both element and listener metadata by splitting on common identifier splitting conventions (e.g., camel casing, underscores, dashes). A part-of-speech tagger tags each sentence, phrase, or individual word, while the system discards non-English strings using an open source spell checker library[3]. The algorithm then searches each string for the first verb, followed by the first noun. If both can be found, the system generates a *command label*.

Genie uses command labels to uniquely identify and trigger commands. The system prioritizes imperative phrases and verbs to use as these labels, but if none can be found will fall back to remaining metadata. Developers can use the remaining metadata to provide a more detailed description of the command to be shown in an interface, such as the one shown in Figure 5. This remaining metadata is labeled as *description metadata.* Thus, the two outputs of this process are a command label and description metadata. In Figure 5, the multiplication command has a command label *Multiply* and description metadata of *Operator button,* and *Ops.*

Invoking Commands

Performing commands through an alternate interface requires some way of triggering the original functionality

[3] github.com/cfinke/Typo.js

Input: mouseMoveHandler AST

```
function mouseMoveHandler(e) {
   var relativeX = e.clientX;
   relativeX = relativeX - canvas.offsetLeft;
   if(relativeX > 0
      && relativeX < canvas.width){
        paddleX = relativeX - paddleWidth/2;
   }
}
```

Output: `True` – Command dependent on mouse position

Figure 6. This event listener references the clientX property of the event object. This is stored in the variable *relativeX* which is referenced in the conditional statement, which guards a side effect. Our system detects these dependencies and determines that the command is dependent upon mouse location.

through the new interface. The JavaScript DOM API provides a `dispatchEvent` method `for` creating and triggering custom events. Genie uses this API to allow Genie interfaces to dispatch events to the original interface, creating a new `Event` object with the necessary inputs, and triggering them through the `dispatchEvent` method defined in the `EventTarget` DOM API.

However, Genie cannot simply trigger only the event corresponding to the command. Web interfaces instead expect one or more sequences of events to be triggered by a motor action (e.g., clicking the left mouse button, typing a key). For example, a `mouseup` event and a `mousedown` event must be triggered before the `click` event is triggered, as an application's semantics may depend on these events occurring. We call these event ordering requirements *device dependency* events (see definition in Figure 2). Each command has a list of *pre* device dependencies and *post* device dependencies describing events that need to be triggered before and after the event, in the correct order. Device dependencies consist of both *direct* events, such as `mouseup` and `mousedown`, and *indirect* events, such as `blur` and `focus`. When a Genie interface requests to perform a command, Genie executes pre and post device dependencies before and after a command in the correct order if there are commands corresponding to those events.

Many events also require additional input that originates from device specific input, such as a mouse location, or key code. Genie analyzes and discovers these dependencies through a command's event listener. To do this, Genie traverses the AST of the event listener to locate mouse location and keyboard dependencies. Event listeners typically reference device location through properties on the event object (e.g., `evt.clientX`, `evt.clientY`, `evt.x`, and `evt.y`). Our algorithm searches the AST for references to these properties, typically in assignment or conditional expressions. For example, Figure 6 shows that `mouseMoveHandler()` references the `clientX` property of the event object and stores it in the `relativeX` variable. The conditional test expression then references the variable `relativeX`. We transitively detect any dependencies that

Input: keyDownHandler AST

```
function keyDownHandler(e) {
     if(e.keyCode == 13) {
          submitOnEnter();
     }
}
```

Output: `KeyCode: 13, submitOnEnter()`

Figure 7. This event listener references the keyCode property of the event object and compares it to the value. Genie returns the value 13 and the corresponding side effect.

we can statically determine will effect the control flow through the event listener.

Genie detects keyboard dependencies similarly (Figure 7), looking for references in the AST to `code`, `key`, or `keyCode` properties on the event object. If the key code of the event is assigned to a variable that is referenced on a conditional expression, or if the key code is referenced directly, we collect the corresponding value that the `keyCode` is compared to (e.g., `if(evt.keyCode == 13)`). If the keyCode value is not hard-coded, Genie transitively determines the value, if possible.

As Figure 2 shows, each command has a *required input* property. Each key code value we discover (e.g., `13`) is added to the *required input* list. If we cannot find a possible value for a key code reference, in cases where the key code value is assigned to a global variable or variable declared outside the function, and not referenced later in a conditional, we do not add a value to *required input* because we cannot determine that value statically. Genie detects mouse button dependencies through traversing references to the `button` and `buttons` properties of the event object, and collects the corresponding values in a similar manner to `keyCode`.

Each key code or mouse button value in the required input property is mapped to one or more side effects. When a command has multiple required input values, Genie splits the command into multiple *pseudo-commands,* where each required input value is a command, has a command label corresponding to its input value or side effect metadata, and has a set of side effects that will occur if the command is given that specific input. Figure 1 is an example where the commands *"Left"* and *"Right"* originate from the same event listener. In the Genie system, a pseudo-command is represented in the same way as a regular command.

Each Genie command also has a *perform* method that triggers the command, supplying the *required input* and *pre* and *post device dependencies.* For a pseudo-command, Genie supplies the associated *required input* value.

Genie API

To support developers in building applications with Genie models, we built an API that exposes any web page's current set of commands, properties, and behaviors. Developers can create a Genie interface which can subscribe to an abstract list of commands that Genie keeps up to date. Genie notifies each interface when the state of a property (e.g., visible,

enabled) has changed, or when a command is added or removed. Each Genie interface requires defining a new representation of the interface to use for a command (i.e., HTML structure and CSS), and code that defines how to update an interface when a command's enabled or visible status changes. However, a Genie interface does not need to define any code that interacts with the abstract set of commands.

Each Genie interface can trigger the command using the framework and pass in the required arguments. Interfaces can define their custom command triggers for each command, which Genie then invokes automatically. Genie provides each interface a set of default labels for a command, including a command trigger label that is unique to each command (e.g., the bolded labels shown in Figure 1) and an additional set of labeling metadata. The framework is meant to be simple, only requiring a new Genie interface to implement the behaviors required for command activation.

GENIE INTERFACE EXAMPLES

The benefit of having an abstract model of application commands is that we can easily translate web interfaces to support a range of input, without having to design built-in support for this range of input. We validated Genie by building several diverse applications showcasing these translations, and we motivate them through their potential for making the web more powerful and accessible. Each of these applications is demonstrated on a single website, but the applications themselves are generic and are meant to be applied to any website based on Genie's analyses.

Automatic Speech Input

Many people have severe motor impairments that make using a mouse or physical keyboard almost impossible [13]. However, many people with motor impairments can use speech interfaces. Using the Genie API, we built the speech interface shown in Figure 1. The Genie interface displays a list of currently available commands on the page. People can trigger any of the commands by simply speaking the label shown in bold. Commands shown in dark grey are currently disabled and cannot be triggered. Genie monitors and updates the states of these commands as the user interacts with the web page. We implemented this interface using the Genie API, defining the interface structure and styles, integrating the Web Speech API[4] to process speech input, and mapping each speech input to the corresponding Genie command. This only required about 150 lines of JavaScript code.

The web site shown in Figure 1 demonstrates our speech interface active on a game called Hextris, which consists of a rotating hexagon. The objective of the game is to prevent blocks from leaving the outside of the gray hexagon. The two main commands to play the game are speaking "Left" and "Right" to rotate the hexagon in either direction. Genie allows anyone to play this game via speech, in addition to using the built-in mouse and keyboard commands.

Automatically Generated Keyboard Shortcuts

Most websites only support mouse input, or selectively implement support for a small set of keyboard accelerators. For example, the calculator shown in Figure 5 is a basic calculator with simple number, operator (e.g., +, *), and memory functions (e.g., MR, MC). It has keyboard shortcuts for numbers, but not for operators or memory functions, rendering the calculator useless without a mouse. Such applications are less accessible to people who do not use a mouse, such as people with severe motor impairments or people who use screen readers.

We used Genie to build the interface shown in Figure 5. This interface displays a list of commands and a corresponding automatically generated keyboard shortcut for each command. Implementing this interface required defining shortcut triggers using the Keypress API[5] and defining a function to generate a shortcut for each command. We defined a simple method to define shortcuts using the first letter of the command label and the modifier `ctrl`. If a shortcut is already used, the interface assigns the second letter as the shortcut, and so on. Each keyboard shortcut has a unique command label and a corresponding metadata description. This required about 130 lines of code.

Figure 5 shows the calculator interface with our Genie interface active. This interface can be used on any website to activate automatically generated shortcuts, so the calculator interface is used here as just an example. The interface displays a shortcut that can be used to trigger each number, operator, and memory command. Number commands contain the metadata "number button", operators have the metadata "operator button", and memory buttons have the metadata "memory button", along with other collected metadata, shown by each command description. For this interface, the labels happen to originate from the referenced listener name in the `onclick` attribute which for each of the buttons has the value `onclick="operatorButton('+')"`.

Keyboard-Based Mouse Input

Many people cannot use a mouse to interact with the web, preventing them from using applications that rely on fine-grained pointer input such as drawing or diagramming tools [13,19]. With Genie, we can easily create alternate ways of supplying precise pointer input using the keyboard.

Figure 8 shows an automatic keyboard-based mouse input application we built currently active on a graph drawing application. Typing `ctrl-i` triggers the command "Insert node" as labeled in the figure. Triggering the "Insert node" command displays a grid covering the surface of the canvas. Typing in the numbers corresponding to the desired cell generates a mouse location used as input to Genie when the command is triggered. Implementing this Genie interface using the framework simply required implementing the method of inputting coordinates, integrating the Canvas API

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

[5] *dmauro.github.io/Keypress*

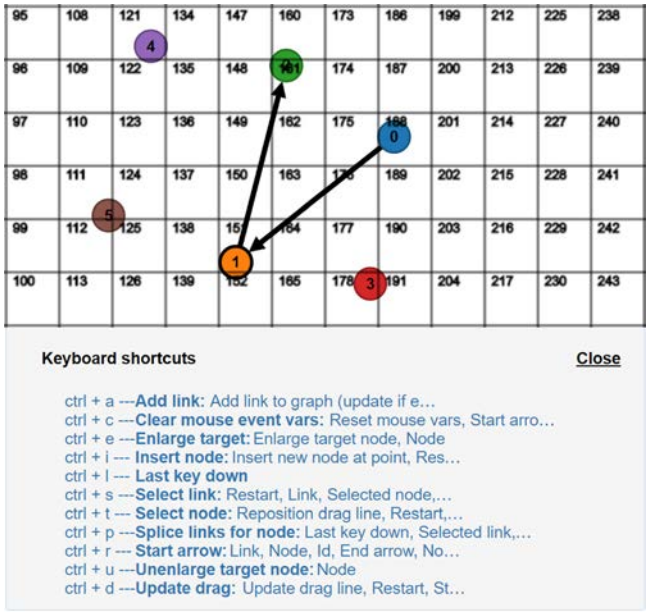

Figure 8. A graph builder augmented with Genie's input grid for capturing mouse coordinates via keyboard.

to draw the grid, and registering `keydown` listeners to process location input. In all, this interface required only about 75 lines of JavaScript.

This method of input, while not having the precision of clicking a mouse on the canvas, does provide a method of entering this input. This interface could easily be extended with more accurate methods of input, such as an onscreen cursor that could be moved with keyboard or voice commands. This interface could also work with other input devices. For example, people could speak the cell numbers to input the location or use the command line interface that we describe in the next section. The developer of each Genie interface would need to create this mapping, but as we have shown, such mappings require very little implementation work.

A Command Line Interface for Web Automation

Most web applications are not scriptable. Many web forms require painstaking input needing a skilled programmer to automate. Web automation tools such as CoScripter [22] and Chickenfoot [5] provided powerful solutions to this problem. Genie can easily recreate such functionality, providing a command line scripting console for arbitrary web applications.

Figure 9 shows our example command-line applied to a simple to-do list application (*flask.io)*. This application allows people to create to-do lists, save to-do lists to a profile, and share tasks with other people. Typing "commands" into the terminal presents the user with a list of the available commands discovered by Genie. Commands for the application shown include "write task" which corresponds to typing a value into the field labeled "Write your next task here", and "save task" which saves the task to the interface. Typing "help" into the terminal displays the list of command triggers along with more detailed descriptions from the collected command metadata. The command line interface

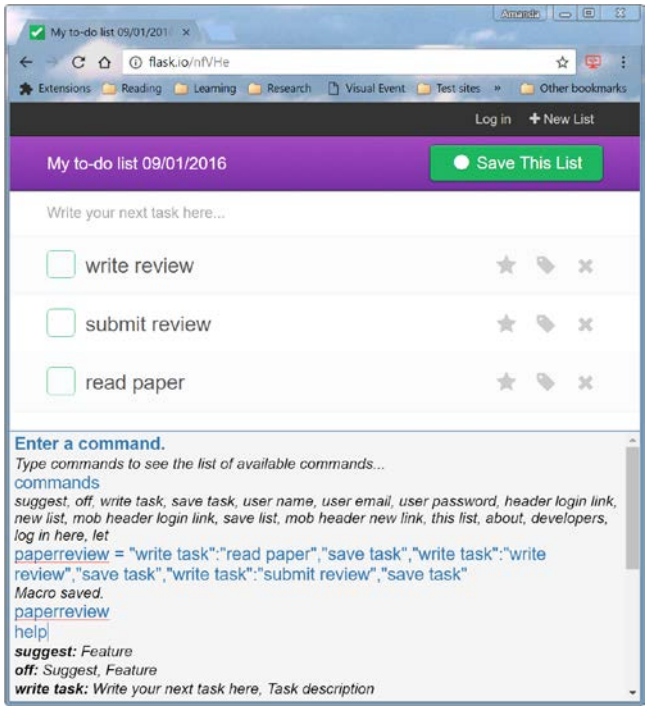

Figure 9. A Genie-enabled command line terminal that allows command automation and macro creation.

supports macro creation allowing for automation of multiple commands and inputs. The user creates a macro using the following format.

```
<macroName>="<commandName1>":"<commandInput1 (optional)>","<commandName2>"…
```

The `paperReview` macro in Figure 9 creates and saves three tasks to the list: *read paper*, *write review*, and *submit review*. These are three hypothetical tasks that a reviewer might create to remember to complete all steps of submitting a paper review. The macro could potentially be persisted across sessions so that adding subsequent paper reviews would require simply typing the command `paperReview`.

RELATED WORK

Prior work has explored various aspects of the Genie system, from enabling creation of more personalized and accessible websites to retargeting the inputs of an application from one modality to another. Fewer systems have explored discovering the interactive behaviors or models of an application and modifying the discovered behaviors. Genie goes beyond such work by integrating program analysis techniques to enable command discovery and customization.

One prior focus of web personalization is web accessibility. ARIA [10] attributes enable screen readers to interpret web content, and a few prior works have dynamically analyzed and injected ARIA attributes into a website. One method describes dynamic updates by monitoring and dynamically injecting ARIA attributes onto the updated content [6]. Another method detects and makes static content accessible [7]. However, these methods primarily operate within the existing input modality of the page, improving the interaction for people

who could already access the page via that modality, but not enabling access via entirely new modalities.

In addition to ARIA attributes, some other approaches have used the power of crowdsourcing and collaboration to identify web accessibility issues and apply fixes, including AccessMonkey [3] and CAN [18]. These systems enable developers to write scripts to fix specific accessibility issues. However, these scripts are mostly written for a specific website or subset of websites. They typically modify a specific aspect of behavior or add a new functionality. In contrast, Genie focuses on a generic method for discovering and describing existing functionality to support alternates forms of access to that existing functionality.

Prior work also explores automatically generating interfaces to make them more accessible or efficient to use. SUPPLE [14] had users take a one-time performance test that enabled generation of a custom interface suited to personal abilities, improving efficiency of generated interfaces. The EKOGI system [1] accounted for a person's abilities and the interactions that best suit those abilities, generating a tailored interface. The contributions of Genie could allow such solutions to leverage a more application-agnostic model of commands. Genie could notify such systems what commands are available, and a corresponding Genie interface could be created per each person's detected abilities. Additionally, these interfaces could potentially be even more personalized because they could select and use an alternate input modality that is more appropriate to a person's abilities.

In addition to Genie, a few systems have enabled input retargeting to translate the modality of an application to one that is more accessible or efficient to use, but these systems have hard-coded input mappings. Gesture Avatar [24] allowed people to interact with an existing mobile interface through gestures. Their method operated on the pixel-level, creating a mapping between gestures and their corresponding objects in the interface. Their approach creates custom mappings from one input domain to another, while Genie allows for the creation of generic mappings.

Genie relies on the ability to automatically discover interactive behaviors of an application. Prior work in program analysis has discovered these interactive behaviors, but has primarily focused such efforts toward GUI testing. For example, some prior work has utilized web crawling to produce models of the interactive components of web interfaces [25,29]. For desktop user interfaces, some prior work has discovered these interactive components using accessibility APIs [27] and computer vision techniques [15].

A few systems have both detected and enabled modifying the behavior of the interactive components of an application. Runtime toolkit overloading in Scotty [12] is one approach, a technique for supporting *manual* program analysis for adding functionality to existing runtime behavior. Prefab [11], Sikuli [32], and Waken [2] all use pixel-based analyses to discover interface components through templates and machine vision techniques. These systems also enable modifying the behavior of the detected interface components. Because these methods only have access to the pixel-level appearance of an application's interface, but not the application's source code, they can only understand visible behaviors. Unlike Genie, they have no understanding about whether components they are detecting are actually interactive, nor any way of predicting their behavior.

Only a few prior works have analyzed source code for enhancing accessibility or usability. Ko et al. applied program analysis to detect paths in a web application that did not result in any feedback in the interface [21]. Genie builds upon these techniques, using both static and dynamic analysis, to support automatic interface translation and adaptation.

LIMITATIONS

Many of Genie's limitations are due to limited availability of metadata in an application's source code. For instance, a key limitation of Genie's command labels and the descriptions Genie discovers is that many websites use "minification" to improve performance and obfuscate code, limiting the information Genie can extract and present in alternate interfaces. Even if a site is not minified, many websites do not include descriptors in the comments or source code of their event listeners or on their command elements. Even if the information exposed by Genie is not detailed or high quality, the ability of Genie to at least expose what actions are possible may still be useful. Better metadata might also be attached through social annotation techniques [20], applied to the original interface or to Genie's representation.

Other limitations are due to imprecision in the program analyses that we used in our prototype. For example, we only analyze the functions registered as event listeners, and not the functions they transitively call. We could have done a full program analysis, tracking the full extent of downstream side effects following a function call, potentially discovering a more precise set of command states (e.g., enabled, disabled), side effects, and descriptions. Not having these precise states might mean that a command label is not descriptive enough, or that a command that is currently disabled is shown as enabled. However, triggering the command through Genie will have the same behavior as triggering the command through the regular interface, and relevant feedback of the disabled command will still be presented. Discovering more precise command states is a matter of applying more advanced program analysis techniques from prior work in software engineering, but it was outside the scope of our prototyping.

Another limitation we discovered with larger websites was that Genie discovered large numbers of commands that made it difficult to discern which command triggered which functionality. Future work should explore reverse engineering more metadata to help organize and group relevant commands together so that they are more discoverable.

DISCUSSION & FUTURE WORK

During the process of creating our alternate interfaces, we generated several additional ideas for Genie interfaces. Many of these involved creating input retargeting support for other modalities, including touch input, brain-computer input, or any other types of future input devices. However, we also generated several other interface enhancements that go beyond input. For example, Genie might be used to automatically create a help interface that display tooltips that describe each command and its side effects. In investigating this, we found that the metadata we could collect from the real applications used in this paper was not detailed enough to support this type of description, but would still give some indication as to the behavior of the command and could be useful in many cases. More advanced analysis of command side effects might allow us to generate tutorials that suggest a specific sequence of commands to complete a particular task, thus providing more advanced and automated help.

Another promising use for Genie might be in adding enhanced ARIA metadata to existing websites, which is a standard for making interactive websites screen readable. For example, the attribute `aria-disabled` indicates that an element is perceivable but not interactive. Hidden elements in a page should be marked with the attribute `aria-hidden` which indicates that the interactive element is not visible or perceivable. As the Genie data model already exposes and notifies a Genie interface when these two properties are updated for any command, it would be simple to implement a Genie interface that keeps these two properties up to date for any web interface. In fact, utilizing a combination of static and dynamic analysis has the potential of being able to monitor the state of many attributes, such as `aria-invalid`, `aria-expanded`, and others. We will explore extending Genie to monitor additional properties in future work.

There are a diverse set of issues with existing web interfaces that could also be enhanced or repaired with the metadata Genie collects. We have explored building a Genie interface that would automatically detect and repair usability issues in an interface. For example, previous work [21] analyzed a set of 115 web applications, and found 37% did not provide feedback to users after completing an action. Augmenting Genie to detect when a command does not provide feedback, and generating customized feedback messages, could be a promising application. Genie additionally analyzes the disabled state of commands, so using this information to provide reasoning to users about why a command is currently disabled, or what commands can be performed to enable it, could also be useful applications of Genie's analysis.

In future work, we plan to scale up our evaluation to demonstrate the effectiveness of our techniques on a larger set of websites selected from the Alexa.com rankings. This will allow us to fully evaluate the benefits and limitations of Genie across a more representative set of websites, and discover areas of improvement for future work.

CONCLUSION

This paper has presented Genie, a framework to reverse engineer the interactive commands from a website, retargeting their inputs to alternate input modalities. Genie enables alternate access to a broad range of websites that were not designed for diverse abilities. By implementing a set of alternate interfaces using the Genie framework, we have shown that this approach has the potential to create more efficient and customizable interfaces that can enhance the ability to interact with existing websites. Through the many opportunities to enhance the metadata that Genie collects, and through more advanced methods of program analysis, we hope to create a system where every website can be accessed by any person that desires to use it, through any method of interaction they require based on their individual abilities.

ACKNOWLEDGMENTS

We thank Dastyni Loksa, Alex Rowell, William Menten-Weil, and Dakota Miller for early feedback on the work and prototypes. We also thank Annie Ross for reading and providing feedback. This material is based upon work supported in part by the National Science Foundation under awards CCF-1153625, IIS-1053868, and IIS-1314399.

REFERENCES

1. Julio Abascal, Amaia Aizpurua, Idoia Cearreta, Borja Gamecho, Nestor Garay-Vitoria, and Raúl Miñón. 2011. Automatically Generating Tailored Accessible User Interfaces for Ubiquitous Services. In *Proceedings of the International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '11)*, 187–194. http://doi.org/10.1145/2049536.2049570
2. Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2012. Waken: Reverse Engineering Usage Information and Interface Structure From Software Videos. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '12)*, ACM Press, 83–92. http://doi.org/10.1145/2380116.2380129
3. Jeffrey P. Bigham and Richard E. Ladner. 2007. Accessmonkey: A Collaborative Scripting Framework for Web Users and Developers. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility (W4A '07)*, ACM Press, 25–34. http://doi.org/10.1145/1243441.1243452
4. Jeffrey P. Bigham, Tessa Lau, and Jeffrey Nichols. 2008. Trailblazer: Enabling Blind Users to Blaze Trails Through the Web. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI '09)*, ACM Press, 177-186. http://doi.org/10.1145/1502650.1502677

5. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the Annual ACM Symposium on User interface Software and Technology (UIST '05)*, ACM Press, 163–172. http://doi.org/10.1145/1095034.1095062
6. Yevgen Borodin, Jeffrey P. Bigham, Rohit Raman, and I. V. Ramakrishnan. 2008. What's new?: Making Web Page Updates Accessible. In *Proceedings of the International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '08)*, ACM Press, 145–152. http://doi.org/10.1145/1414471.1414499
7. Andy Brown and Simon Harper. 2013. Dynamic Injection of WAI-ARIA into Web Content. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility (W4A '13)*, ACM Press, Article 14. http://doi.org/10.1145/2461121.2461141
8. Ben Caldwell, Michael Cooper, Loretta Guarino Reid, Gregg Vanderheiden. 2008. Web Content Accessibility Guidelines (WCAG) 2.0. Retrieved September 18, 2016 from https://www.w3.org/TR/WCAG20/
9. Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 2011. Associating the Visual Representation of User Interfaces with their Internal Structures and Metadata. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '11)*, ACM Press, 245-256. http://doi.org/10.1145/2047196.2047228
10. Michael Cooper. 2016. WAI-ARIA Overview. Retrieved September 18, 2016 from https://www.w3.org/WAI/intro/aria
11. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*, ACM Press, 1525–1534. http://doi.org/10.1145/1753326.1753554
12. James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '11)*, 225–234. http://doi.org/10.1145/2047196.2047226
13. Leah Findlater, Alex Jansen, Kristen Shinohara, et al. 2010. Enhanced Area Cursors: Reducing Fine Pointing Demands for People with Motor Impairments. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '10)*, ACM Press, 153–162. http://doi.org/10.1145/1866029.1866055
14. Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '07)*, ACM Press, 231–240. http://doi.org/10.1145/1294211.1294253
15. Paul Givens, Aleksandar Chakarov, Sriram Sankaranarayanan, and Tom Yeh. 2013. Exploring the Internal State of User Interfaces by Combining Computer Vision Techniques with Grammatical Inference. In *Proceedings of the International Conference on Software Engineering (ICSE '13)*, IEEE Press, 1165–1168. Retrieved from http://dl.acm.org/citation.cfm?id=2486951
16. Vicki L. Hanson and John T. Richards. 2013. Progress on Website Accessibility? *ACM Transactions on the Web* 7, 1: 1–30. http://doi.org/10.1145/2435215.2435217
17. Andreas Holzinger, Gig Searle, and Alexander Nischelwitzer. 2007. On Some Aspects of Improving Mobile Applications for the Elderly. In *International Conference on Universal Access in Human-Computer Interaction*, 923–932. http://doi.org/10.1007/978-3-540-73279-2_103
18. Yun Huang, Brian Dobreski, Bijay Bhaskar Deo, et al. 2015. CAN: Composable Accessibility Infrastructure via Data-Driven Crowdsourcing. In *Proceedings of the Web for All Conference (W4A '15)*, ACM Press, Article 2. http://doi.org/10.1145/2745555.2746651
19. Faustina Hwang, Simeon Keates, Patrick Langdon. 2004. Mouse Movements of Motion-Impaired users: A Submovement Analysis. In *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '04)*, 102–109. http://doi.org/10.1145/1028630.1028649
20. Shinya Kawanaka, Yevgen Borodin, Jeffrey P. Bigham, Darren Lunn, Hironobu Takagi, and Chieko Asakawa. 2008. Accessibility Commons: A Metadata Infrastructure for Web Accessibility. In *Proceedings of the International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '08)*, 153–160. http://doi.org/10.1145/1414471.1414500
21. Amy J. Ko and Xing Zhang. 2011. Feedlack Detects Missing Feedback in Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, ACM Press, 2177–2186. http://doi.org/10.1145/1978942.1979260

22. Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*, ACM Press, 943-946. http://doi.org/10.1145/1240624.1240767
23. Tovi Grossman and Ravin Balakrishnan. 2005. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*, 281–290. http://doi.org/10.1145/1054972.1055012
24. Hao Lü and Yang Li. 2011. Gesture Avatar: A Technique for Operating Mobile User Interfaces Using Gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, 207–216. http://doi.org/10.1145/1978942.1978972
25. Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2012. ReAjax: A Reverse Engineering Tool for Ajax Web Applications. *IET Software* 6, 1: 33–49. http://doi.org/10.1049/iet-sen.2010.0152
26. Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: An Innovative Tool for Automated Testing of GUI-Driven Software. *Automated Software Engineering* 21, 1: 65–105. http://doi.org/10.1007/s10515-013-0128-9
27. Christopher Power, André Freire, Helen Petrie, and David Swallow. 2012. Guidelines Are Only Half of the Story: Accessibility Problems Encountered by Blind Users on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*, ACM Press, 433–442. http://doi.org/10.1145/2207676.2207736
28. Katharina Reinecke and Abraham Bernstein. 2011. Improving Performance, Perceived Usability, and Aesthetics with Culturally Adaptive User Interfaces. *ACM Transactions on Computer-Human Interaction (ToCHI '11),* 18, 2: 1–29. http://doi.org/10.1145/1970378.1970382
29. Carlos E. Silva and José C. Campos. 2013. Combining Static and Dynamic Analysis for the Reverse Engineering of Web Applications. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, 107–112. http://doi.org/10.1145/2494603.2480324
30. Steven John Simon and Steven John. 2001. The Impact of Culture and Gender on Web Sites. *ACM SIGMIS Database: The Database for Advances in Information Systems* 32, 1: 18–37. http://doi.org/10.1145/506740.506744
31. WebAIM: Web Accessibility in Mind. 2015. WebAIM: Screen Reader User Survey #6 Results. Retrieved September 18, 2016 from http://webaim.org/projects/screenreadersurvey6/
32. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST '09)*, 183–192. http://doi.org/10.1145/1622176.1622213