Enhancing Cross-Device Interaction Scripting with Interactive Illustrations

Pei-Yu (Peggy) Chi[1,2 *], **Yang Li**[1], **Björn Hartmann**[1,2 *]
Google Inc.[1]
Computer Science Division, UC Berkeley[2]
peggychi@cs.berkeley.edu, yangli@acm.org, bjoern@cs.berkeley.edu

ABSTRACT
Cross-device interactions involve input and output on multiple computing devices. Implementing and reasoning about interactions that cover multiple devices with a diversity of form factors and capabilities can be complex. To assist developers in programming cross-device interactions, we created DemoScript, a technique that automatically analyzes a cross-device interaction program while it is being written. DemoScript visually illustrates the step-by-step execution of a selected portion or the entire program with a novel, automatically generated *cross-device storyboard* visualization. In addition to helping developers understand the behavior of the program, DemoScript also allows developers to revise their program by interactively manipulating the cross-device storyboard. We evaluated DemoScript with 8 professional programmers and found that DemoScript significantly improved development efficiency by helping developers interpret and manage cross-device interaction; it also encourages testing to think through the script in a development process.

Author Keywords
Cross-device interaction; scripting; interactive illustration; storyboards.

ACM Classification Keywords
H.5.2. User Interfaces — prototyping, input devices and strategies, graphical user interfaces.

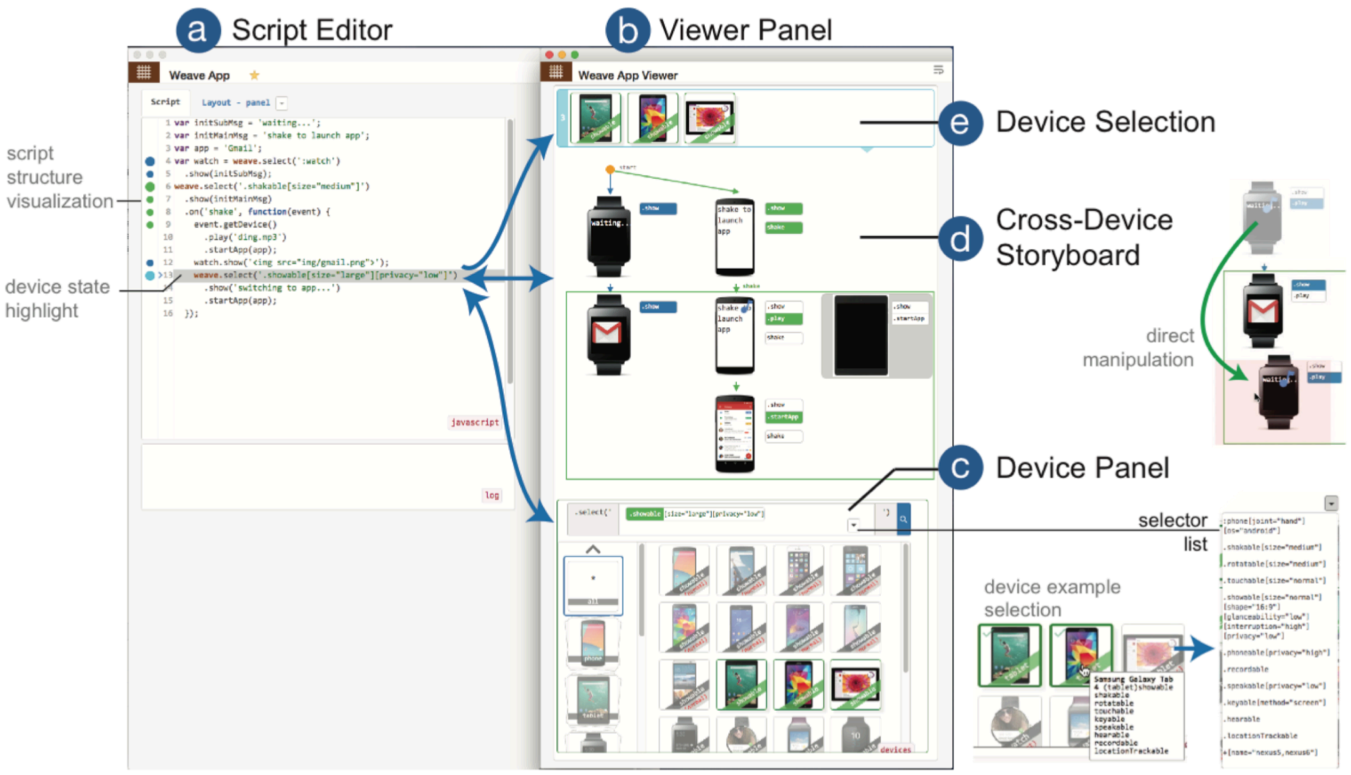
INTRODUCTION
Wearable and mobile devices have introduced new ways for interaction with their unique capabilities and form factors. As consumers embrace a multi-device ecosystem, interactions spanning multiple devices can bring significant benefits beyond operating single devices. Examples that have been shown include: cross-display content manipulation [29, 32, 37], watch-phone interactions [6, 11, 16], multiple cube-size device gaming [34], and interactive augmented space via smart eyewear [10, 27].

Implementing cross-device interactions poses several challenges to developers. First, *it is difficult to account for the wide variety of form factors and capabilities* of devices *and their combinations* that a program may be run on. For example, a cross-device camera app may have to provide several versions to offer similar interactions on a phone and a watch or an eyewear for consistency. Second, *it is challenging to express and reason interaction flows that span multiple devices*. The higher-level relationships between cross-device inputs and outputs are often defined in a variety of callback functions spread across a codebase. Third, *testing cross-device interactions remains difficult*, especially when various physical inputs across devices are involved. Developers often have to resort to stepping through applications and manually provide input to test these interactions.

Prior work has aimed to lower the complexity in cross-device development through programming frameworks or toolkits by abstractions. Panelrama introduced a constraint-based approach for specifying cross-device UIs [37]. Weave's scripting framework allows developers to create cross-device behaviors using a high-level API inspired by Web programming models [7]. WatchConnect provides a platform for prototyping watch-centric apps considering watch-specific capabilities [16]. In this paper, we take a complementary but independent approach: We hypothesize that a key difficulty lies in bridging the gap between envisioning concrete examples of interactions and abstracting specification in code. When designing cross-device interactions, developers have to translate device-specific scenarios into abstract specifications to a range of devices at runtime. *The gulf of execution* [30, 25]—writing device selection and event handler statements—and *the gulf of evaluation*—reasoning whether existing statements specify the correct set of devices and interactions—can be significant when number of devices and form factors increase. Our goal is to minimize these gulfs by 1) illustrating the behavior of code with concrete examples,

CHI'16, May 07-12, 2016, San Jose, CA, USA
ACM 978-1-4503-3362-7/16/05.
http://dx.doi.org/10.1145/2858036.2858382

* This work was done while the first and the third authors were respectively an intern and an academic consultant at Google.



```javascript
var initSubMsg = 'waiting...';
var initMainMsg = 'shake to launch app';
var app = 'Gmail';
var watch = weave.select(':watch')
  .show(initSubMsg);
weave.select('.shakable[size="medium"]')
  .show(initMainMsg)
  .on('shake', function(event) {
    event.getDevice()
      .play('ding.mp3')
      .startApp(app);
    watch.show('<img src="img/gmail.png">');
    weave.select('.showable[size="large"][privacy="low"]')
      .show('switching to app...')
      .startApp(app);
});
```

```
:phone[joint="hand"][os="android"]
.shakable[size="medium"]
.rotatable[size="medium"]
.touchable[size="normal"]
.showable[size="normal"][shape="16:9"][glanceability="low"][interruption="high"][privacy="low"]
.phoneable[privacy="high"]
.recordable
.speakable[privacy="low"]
.keyable[method="screen"]
.hearable
.locationTrackable
+[name="nexus5,nexus6"]
```

```
Samsung Galaxy Tab 4 (tablet)showable
shakable
rotatable
touchable
keyable
speakable
hearable
recordable
locationTrackable
```

Figure 1. DemoScript allows developers to script the abstract logic in the Script Editor (a) and in real-time observe and manipulate the simulation in the Viewer (b). Developers can specify device examples in the Device Panel (c), observe the device selection (e), and verify the application logic in the cross-device storyboard (d). The storyboard employs a grid layout where each column represents a device selection, and each row shows the device state progress in time connected by arrows to show the interaction flow.

and 2) the opposite direction—generating code from direct manipulation of the visual illustration.

We present DemoScript, a novel technique that assists developers in programming cross-device interaction by interactively illustrating the behavior of the program. Based on an understanding of the syntax and semantics of a cross-device UI framework, DemoScript analyzes the program as the developer enters it, and in real-time automatically generates and presents a visual illustration of the scripted behavior as a cross-device storyboard (see Figure 1).

In our approach, developers move between authoring code (*i.e.*, abstract application logic) and manipulating visualizations (*i.e.*, examples of concrete run-time executions) of aspects of their code. The visualizations focus on key aspects of cross-device interactions—device selection and interaction flow—and show examples of how the written behaviors give rise to different behaviors across multiple devices. Developers can navigate in code line-by-line to see the partial, step-by-step execution, which helps them connect the abstract code to run-time behaviors. Through the cross-device storyboard, developers can revise their program for a range of aspects by directly manipulating the elements in the storyboard. Our approach effectively encourages testing and visually verifying cross-device interactions that involve various device form factors and interaction capabilities.

An evaluation of DemoScript with 8 professional programmers indicated that our technique significantly improves the efficiency for programming complex interaction behaviors such as cross-device interaction, in comparison with a baseline approach. In particular, our work makes the following contributions:

- A novel approach for programming cross-device interactions, based on a seamless coupling of scripting and interactive visual illustration of scripted behaviors;
- Interactive cross-device storyboards—a novel visualization of interaction logic, and a set of initial findings on the usefulness and usability of cross-device storyboards. Storyboards allow developers to revise a script by direct manipulation and visualizing code fragments as well as entire programs;
- A set of methods for analyzing a cross-device interaction script in a given UI framework.

RELATED WORK

Our work touches on three research topics: enhanced code editors, cross-device IDEs, storyboards in interaction design, and iterating with examples. We here discuss how our work is related to each of these areas.

Enhanced Code Editors

Tools that visualize different aspects of source code and allow direct manipulation of abstractions can successfully lower the complexity of programming [28]. Prior work has attempted to integrate many software visualization techniques into code editors [8]. For example, Codelets shows interactive examples and documentation inside an IDE [31]. Stacksplorer visualizes the structure of call graphs [18]. Beyond enhancing IDEs for general programming tasks, research has also focused specifically on techniques for supporting the development of interactive applications. Most IDEs nowadays ship with a GUI builder to enable developers to graphically layout user interfaces. Xcode further allows developers to create UI sequences for Apple Watch and interactively monitor variables in a program [2, 3]. Juxtapose allows programmers to explore interface alternatives [14] using a linked editing technique [35], and DejaVu helps programmers understand computer vision-based applications that extract interaction logic from camera input [19].

Similar to prior work, we intend to enhance program editors for interactive systems. However, we specifically focus on the unique challenges emerged from cross-device interaction development. We designed DemoScript based on a fundamental understanding of programming frameworks for cross-device interactions.

Cross-Device Interaction

In multi-device research, authoring support tends to focus either on graphical, direct manipulation authoring, or on programming frameworks without visual editors. In the first category, Damask provides a GUI for designers to specify UI patterns across devices [26], while XDStudio supports authoring distributed UIs visually on emulators or actual devices [29]. In the second category, the WatchConnect toolkit comprises both hardware and software components for rapid prototyping of watch-centric cross-device applications [16]. Panelrama uses a constraint-based approach for specifying cross-device UIs [37]. Our work seeks to find a middle ground between these two approaches. On one hand, we focus on preserving the flexibility and high complexity ceiling of authoring interactions in a general purpose programming language. On the other hand, we contribute interactive storyboards that help developers understand the interactions in code.

Storyboards

Screenflow diagrams or UI storyboards are widely used in practice to help developers visualize the flow and think through the logic of an application [36]. They show a sequence of UI screens, each representing application output triggered by user inputs. Researchers have proposed systems that use storyboards to author various kinds of interactive prototypes without code, including sketch-based GUIs [21], sensor-driven mobile prototypes [13], and activity-based applications [23]. Storyboards can also be generated from user demonstration. For examples, FrameWire extracts interaction logic from a paper prototype walkthrough [24] and the Designers' Outpost captures site maps on a smart whiteboard [20].

In contrast to these approaches, we extend the expressiveness of storyboards by introducing a new format based on the unique aspects of cross-device interaction. In addition, our storyboards are closely coupled with scripting—the changes in a script are reflected in the storyboard instantaneously and vice versa.

Iterating with Examples

Examples help concretize and illustrate abstract concepts and can serve as a guide for exploring design spaces. Research has contributed example-centric systems for using examples to search, filter and explore information spaces. For example, users can look for visual website designs by navigating a corpus of specific examples and indicate preferences such as "show more like these selections" [33, 22]. Other efforts include refining and manipulating database queries interacting directly with example results [1], and improving image search results by learning similar features from user-specified examples [9]. In our work, we employ examples to help developers specify selection queries for devices from a wide range of options.

SCRIPTING WITH DEMOSCRIPT

DemoScript helps developers author and test a cross-device interaction script using interactive illustrations. It automatically analyzes a script as it is entered or modified by a developer, based on a cross-device UI framework—in our current implementation, we use the Weave framework proposed previously [7]. DemoScript identifies code relevant to device specification, user input events, and device actions (UI output), and visualizes the interaction flow for an entire program or a subset of code as a cross-device storyboard and presents it side-by-side along the script editor (see Figure 1).

Our cross-device storyboard illustration is based on a grid layout (see Figure 2f). Each column of the grid represents a selected device, and each row presents a state of the cross-device application that is characterized as the combination of device states (across columns). Each cell in this storyboard thus shows the state of a specific device at a particular point of the script execution. If there is no change on a device at an execution step, its representation is omitted on the row such that the developer can easily spot devices that have state changes. The directional arrows that connect cells are transitions, which visualize the interaction flow from one device state to another. A transition can be triggered by a device action (e.g., showing an image on the display) or a user event (e.g., shaking the device).

To discuss how a developer would benefit from DemoScript in scripting a cross-device behavior, assume a developer, Megan, wants to create a photo sharing application such as Instagram. In her design, a watch shows notifications of incoming photos as thumbnails, and tapping on a thumbnail on the watch will open a larger view of this photo on a device with a larger-form factor device, such as a smartphone or a tablet. As the photo is opened on another device, the watch then switches to the Maps application showing the geographical location of this photo. We demonstrate how Megan uses a scripting tool enhanced by DemoScript. We highlight how our tool helps developers on a set of key tasks in cross-device development.

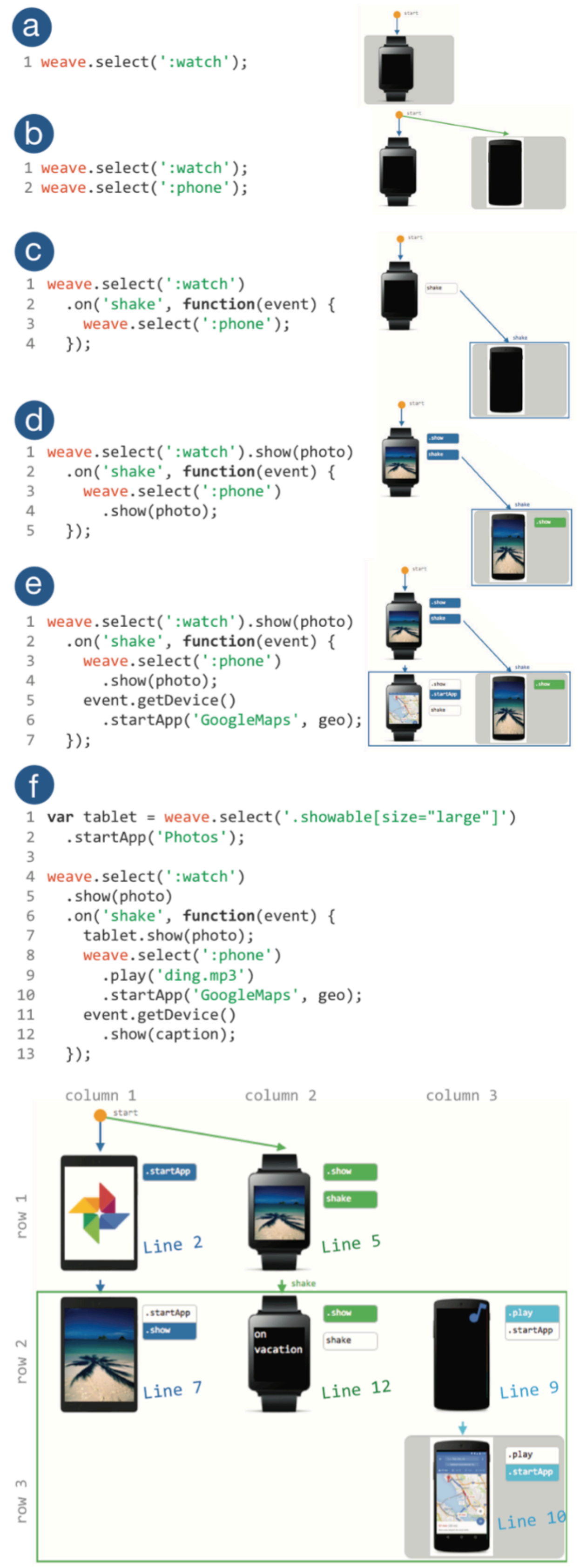
Specifying Target Devices and Transitions

Megan starts by specifying her target devices for this application. To do so, she creates a device selection in the Script Editor (see Figure 1a) by entering `weave.select(':watch')`. As she enters the script, she immediately sees an LG G Watch emulator rendered on the storyboard (see Figure 2a). The top ribbon of the Viewer shows a Device Selection list of four smart watches (see Figure 1e) that match this selector. Megan can choose a target emulator by selecting a device from the list. At the bottom, a Device Panel shows the device repository in which she can browse through possible devices (see Figure 1c). To specify the device for photo viewing, she creates another selection for a phone by entering `weave.select(':phone')` in the Script Editor. Megan sees a phone emulator displayed on the storyboard column next to the watch emulator (see Figure 2b).

Megan then realizes that the phone is needed only when the user shakes the watch. Without manually modifying her script, she adds a transition from the watch to the phone by double-clicking the two device states in sequence in the storyboard and associates the "shake" event to it in the popup. This direct manipulation automatically refactors the script by attaching an event callback to the watch selection and nesting the photo selection within the callback:

```
.on('shake', function(event) {
    weave.select(':phone');
 })
```

Correspondingly, the phone emulator is pushed down a row in the storyboard and connected with a transition from the watch, which indicates the use of watch is triggered by the shake event from the watch (see Figure 2c). To help Megan understand the scope of the callback function (see Line 2-4), DemoScript visualizes the scope as an *event block* that encapsulates any device states that take place *only* in this callback function scope. A transition arrow tagged with an input event is added from the *source* device that triggers the event—the watch that fires the shake event (Line 2)—to the event block.



a

```
1 weave.select(':watch');
```

b

```
1 weave.select(':watch');
2 weave.select(':phone');
```

c

```
1 weave.select(':watch')
2   .on('shake', function(event) {
3     weave.select(':phone');
4   });
```

d

```
1 weave.select(':watch').show(photo)
2   .on('shake', function(event) {
3     weave.select(':phone')
4       .show(photo);
5   });
```

e

```
1 weave.select(':watch').show(photo)
2   .on('shake', function(event) {
3     weave.select(':phone')
4       .show(photo);
5     event.getDevice()
6       .startApp('GoogleMaps', geo);
7   });
```

f

```
1 var tablet = weave.select('.showable[size="large"]')
2   .startApp('Photos');
3
4 weave.select(':watch')
5   .show(photo)
6   .on('shake', function(event) {
7     tablet.show(photo);
8     weave.select(':phone')
9       .play('ding.mp3')
10      .startApp('GoogleMaps', geo);
11    event.getDevice()
12      .show(caption);
13  });
```

Figure 2. An example of scripting cross-device interaction enhanced by DemoScript.

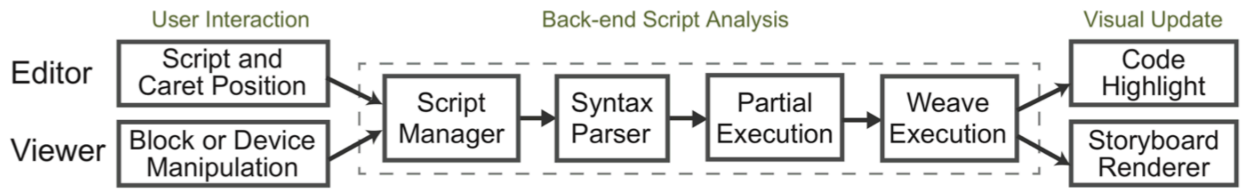


Figure 3. The processing pipeline of DemoScript. The system analyzes developer manipulations and scripting edits in the background to provide real-time, visual feedback in the IDE.

Specifying Device Actions

Next, Megan starts designing the device outputs that users would want to see. To simplify our discussion, assume the program has predefined the resource handle to the photo and its metadata as variables `photo`, `geo`, and `caption` respectively. In the Script Editor, Megan adds `.show(photo)` to the watch selection before the event callback, which lets the watch show a thumbnail of the `photo`. Similarly, she adds `.show(photo)` to the phone selection to show a large view of the `photo` on the phone in response to the shake event. In the process, she immediately sees a sample photo being rendered on both device emulators (see Figure 2d).

While the phone offers a large view of the image, Megan wants the watch to display the photo's geographical location in a Maps application. To do so, in the callback (see Figure 2e) she adds:

```
event.getDevice().startApp('GoogleMaps', geo)
```

With that, the watch switches to a map view the moment when the photo is opened on the phone. All these changes in the script are automatically captured and presented in the storyboard. Both devices are shown on the same row because their states all have changed in response to the shake event (see Figure 2e). Note that the event block has also been expanded to include the device state updates of both the watch and the phone.

Revising Device Selection Criteria with Examples

Megan acquires a good understanding of the look and feel of her choice of devices for the application. From the phone visualization in the storyboard, Megan realizes that users might prefer an even larger view of the photo such that users can easily annotate the photo. To this end, she decides to modify the device selection for the phone, she points to the line where the selector, `weave.select(':phone')`—is declared (see Line 3 in Figure 2e). In response to this action, DemoScript automatically selects the column of the phone selection, updates the Device Selection ribbon at the top of the Viewer Panel, and opens the device repository panel at the bottom (see Figure 1e and 1c).

Megan then selects a few tablets from the device repository, including a Nexus 9 and a Surface Pro. Given these device examples, DemoScript infers a list of selector options based on the attributes of these devices, including device types and capabilities. We will discuss the underlying mechanism in the following section. In particular, DemoScript suggests `':tablet[os="android,windows"]'` as a top choice. After reviewing device examples resulted from this selector in the Device Selection list, she discovers that this selector does not cover laptops that can be converted to a tablet. As such, she explores other suggestions in the selector list offered by DemoScript (see Figure 1c) by testing them out—clicking on each option to view matched device examples. She finally chooses the selector `'.showable[size="large"]'` that better matches her expectation. She then clicks the "Send" button to commit the selector to the script.

Reusing Interaction Behaviors by Drag-and-Drop

After testing her application in the storyboard, Megan decides that showing the map on a phone would be preferable to the watch that has limited display real estate. To do so, instead of modifying the script, she directly drags the map view on the watch to the third, unoccupied column. This direct manipulation creates another instance of the watch selection with the map shown on the third column. Meanwhile, DemoScript creates another selection in the callback function. Because the developer will most likely change the device selection, instead of having two watches in the application, DemoScript automatically prompts the developer to specify a selector. She updates the selector with `':phone'`. Once again, the change is immediately reflected in the storyboard.

Finally, Megan adds more user feedback to enrich this application and sees her complete storyboard as shown in Figure 2f. As the script grows to be more complicated, she could select a block of code to *partially render* the storyboard. For example, selecting Line 7-10 in Figure 2f will re-render the storyboard to show only a tablet showing a photo (Line 7) and a phone playing sound and then launching an app (Line 9-10).

From these examples, we show that DemoScript is closely integrated with script editing. DemoScript enables the developers immediately see the dynamic update on the storyboard based on the position of the script in the Editor. This helps understand the mapping between a portion of code to its runtime execution. Our cross-device storyboard enables a set of visual representations and direct manipulation operations that correspond critical tasks in scripting a cross-device interaction. It also provides an effective visualization of the interaction flow. For example, by looking at how often transitions run across columns, a developer can easily understand the amount of attention shift during an interaction behavior, which is a critical factor of the usability of a cross-device application.

THE DEMOSCRIPT SYSTEM

In this section, we discuss the underlying mechanisms of our system. DemoScript is integrated with a scripting editor and continuously analyzes the script to dynamically generate a cross-device storyboard visualization. Particularly, our implementation analyzes Weave scripts written in JavaScript [15], by obtaining the syntax tree of the code and partially executing it to provide visual assistance in real-time (see Figure 3).

Script Analyses

DemoScript first parses the script into an abstract syntax tree (AST). An AST represents the essential syntactic structure of source code in a tree structure. ASTs have been widely used for program analysis [4] and can be used for program slicing to enable automatic partial testing [38] and potentially usability evaluation [17].

We traverse an AST and recursively build a simplified tree that expresses hierarchical relationships between key constructs of the Weave API, which includes 1) device selectors, 2) actions that change UI state, and 3) events and their handlers that define which input events trigger the actions. *Device selectors* use a declarative query language to define desired properties, similar to CSS selectors for choosing elements on Web pages. For example, `.select('.showable')` selects devices that have a display to show visual outputs. *Actions* are declarations to change the device states, such as updating UI, launching existing application, or playing sounds. An action can be triggered in a callback function that listens to any specific *event* emitted by a device, e.g., touching the screen or shaking the device. DemoScript also identifies and tracks location information for each Weave construct, including its line and column indices to the source code.

Cross-Device Storyboard Generation

DemoScript renders the cross-device storyboard by traversing the AST. It focuses on visualizing device selections and their interaction relations. DemoScript first generates an initial view by rendering emulators for each selection shown in different columns. It then partially executes the code based on the caret position in the Script Editor and dynamically adjusts the layout.

Device Selection Rendering

For each selection statement in the script, DemoScript creates a new column in the storyboard view. To visualize the lifecycles of device selections (*i.e.*, if a device is operated *only* when a certain event is fired), it differentiates the top-level selections $\{selection_{\text{top}}\}$ and those exist in event callbacks $\{selection_{\text{event}}\}$ by presenting them on different rows—pushing $\{selection_{\text{event}}\}$ down one row. For example, in the script in Figure 2f, two new selections via `weave.select` (from Line 1 and 4) are shown in the first and second columns. Inside the *shake* callback function, another new selection (from Line 8) is added to the third column but pushed to the second row.

Next, it is important to visualize the specific "scope" in an interaction flow for developers to visually track the device states before and after an event. Therefore, DemoScript renders an *event-associated arrow* and an *event block* to highlight the relation between devices. To help viewers follow the interaction flow, DemoScript adds a transition arrow from the $selection_i \in \{selection_{top}\}$ that triggers the event i to another $selection_j \in \{selection_{event\_i}\}$ that exists in the callback. The transition arrow is tagged with an event type, such as "shake" or "rotate". It then applies an event block to wrap $\{selection_{event\_i}\}$, *i.e.*, all the selections created in the callback. If there is no new selection, it adds an empty block. The initial view is shown in Figure 4 top-left. The default opacity to these visual elements is set to be low and will be highlighted when the script is executed.

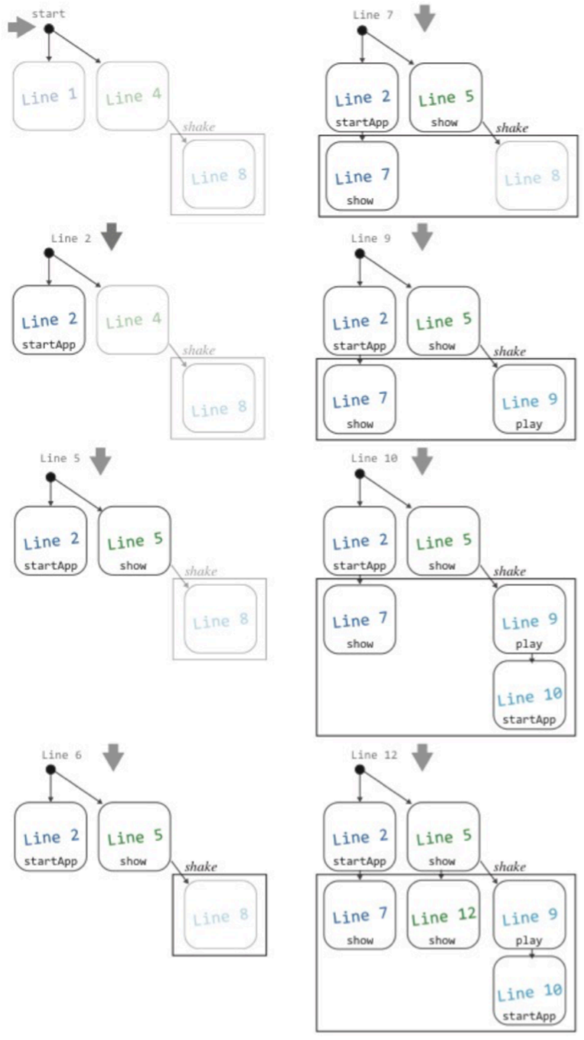


Figure 4. Storyboard adjustments by partial execution of a script based on the developer's navigation in the code.

Code Fragment Execution

When a developer navigates in the script, DemoScript receives the caret position p = ($line_m$, $column_n$) from the Script Editor. It finds the specific element in code at p and identifies all the other Weave elements from the entry point of the script to p. For example, if m = *10* and n is between [7, 36] as `startApp('GoogleMaps', geo)` shown in Figure 2f, the relevant elements include several actions (`'startApp'` in Line 2 and 10, `'show'` in Line 5 and 7, and `'play'` in line 9) and an event (`'shake'` in Line 6) for three different device sets, sorted by closures and positions in code. For each partial execution sequentially from the entry point to p, DemoScript dynamically executes these partial elements and simulates the events in the sequence in the Weave framework. It then adjusts the storyboard (see Figure 4) and inserts the updated UI view for any execution that triggers a user feedback (e.g., showing a message, playing a sound, or starting an app).

Next, DemoScript adjusts the event block to wrap all the UI states triggered by the corresponding callback. For example, when developer navigates to Line 7 that is inside the "shake" callback, a new UI for the watch is appended below the initial view of the device, and the "shake" event block is dynamically expanded to include this view (see Figure 4 top-right). In other words, the developer would only see the partial view of the entire storyboard with all screens created prior to the first execution of Line 7. This interactive method that generates a partial view of the entire script allows developers to easily find the mapping between code and execution. In addition, because DemoScript maps the storyboard and code elements, it enables interactions for developers to directly manipulate device state or selection and update the script automatically.

If the caret is at a selection, it then shows the Device Panel that allows developers to manually specify device examples (see Figure 1e). Any update of the selector will replace the original selector string based on its start and end position retrieved from the AST.

Deriving Device Selection from Examples

DemoScript enables developers to specify a selector by giving examples from a device catalog. Given q device examples [d_1,..., d_q] in a repository of N devices where $q \in [1, N]$, DemoScript adopts a rule-based principle to generate a selector list. Each device d_i with k capabilities is represented as a vector ($cap_i prop_j$, $cap_i prop_{j+1}$, ..., $cap_{i+1} prop_j$, $cap_{i+1} prop_{j+1}$, ..., $cap_k prop_{j+r}$), where *cap* is a device capability for d_i (e.g., `'.showable'` and `'.shakable'`) and *prop* is a property of this capability, such as `'privacy'`, `'glanceability'`, and `'size'`. This device-specific information is presumably defined by manufacturers; we modeled it manually from published device specifications.

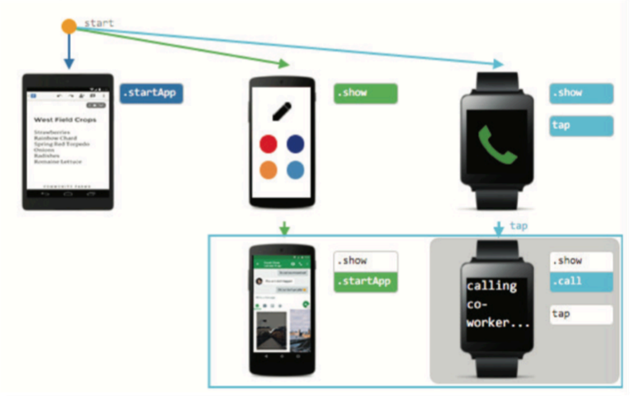
DemoScript generates a list of selectors for each common $cap_i prop_j$ of these capability vectors between the selected devices as: {$selector_1$=`'cap`$_1$`[prop`$_{1\text{-}1}$`="value`$_a$`", prop`$_{1\text{-}2}$`="value`$_b$`"]'`, $selector_2$=`'cap`$_2$`[prop`$_{2\text{-}1}$`="value`$_c$`"]'`, ...}. Each

a

```
// Collaborative editing on a tablet
weave.select(':tablet')
  .startApp('Docs');

// A medium-size device as a sub-panel
var panel = weave.select('.showable[size="medium"]')
  .show(weave.getLayoutById('palette'));

// Call the collaborator via a small device
weave.select('.phoneable[size="small"]')
  .show('<img src="makeCall.png"/>')
  .on('tap', function(event) {
    // call the collaborator
    event.getDevice().call('co-worker');
    // switch the panel to text chat
    panel.startApp('Chat');
  });
```



b

```
var musicFile = 'music.mp3',
  title = 'What A Wonderful World';

// A small device as controller
weave.select('.touchable[size="small"]')
  .show(weave.getLayoutById('volumnControl'));

// Play a song on an Android phone
weave.select(':phone[os="android"]')
  .play(musicFile)
  .show(title)
  .on('longTap', function(event) {
    // look for music videos on a larger device
    weave.select('.showable[size="large"]')
      .startApp('YouTube', title);
  });
```

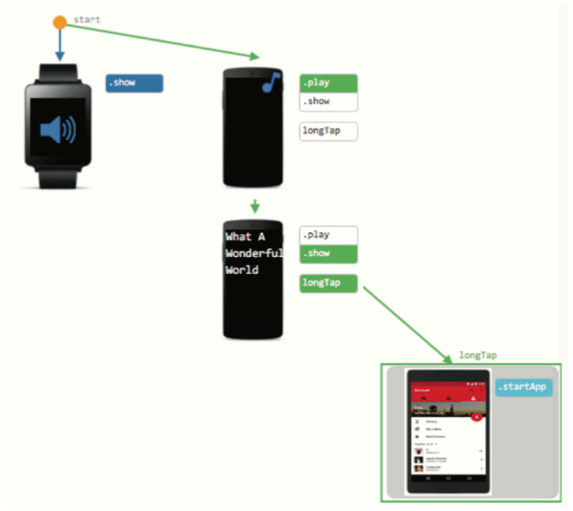


Figure 5. Examples of DemoScript storyboard results.

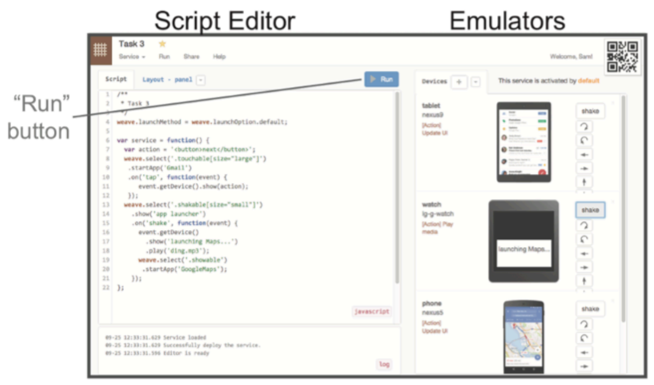
selector$_i$ will match a list of devices in the repository so that the examples [$d_1,\ldots,d_q$] are contained in the device selection. In addition to capabilities, DemoScript also considers selectors by more relaxed—such as using device types, e.g., `':phone[os="android"]'`—or constrained attributes—such as device names, e.g., `'*[name="Nexus5, GalaxyNote4"]'`.

Finally, the list of selector options are ranked based on the number of device matches resulted from each selector in an ascending order, i.e., the more constrained the selector is, the higher it is ranked. In this way, developers can choose a selector based on their coverage and how well the coverage matches their desired devices.

RESULTS
Figure 5 shows two additional examples of the DemoScript storyboard results. The script in Figure 5a assigns one of the devices a different role from an annotation palette (Line 7) to a chat panel (Line 16) when an event happens (*i.e.*, phone calling triggered by the user, Line 12-17). This role change is visually depicted by the second column and the event block. In Figure 5b, the columns show the individual functions of each device (a volume controller, a music player, and a video player respectively) and their progress in time considering device capabilities.

IMPLEMENTATION
The DemoScript system consists of both a backend server and a frontend user interface. Our Node.js-based[1] web server has several functions: 1) it hosts the developers' scripts, 2) runs the Weave framework, 3) stores the device repository in the JSON format, and 4) maintains the DemoScript logic. A user interacts with the frontend application, which includes the web IDE and the Viewer, implemented as a Chrome app [12]. When the developer edits in the Script Editor of the IDE, the front-end app updates the edited script and the current caret position in the code to the back-end server. To analyze the JavaScript code, we integrated an ECMAScript AST parser Esprima [15] into our server to obtain the full syntax tree, which maintains the code hierarchy. If the edited script is executable, the server partially executes Weave code by traversing the AST in order to store information, including device selections and properties. This data is sent in JSON back to the front-end UI, which highlights the code in the Editor, visualizes the storyboard using D3 [5], and handles developer interaction via jQuery[2] and Bootstrap[3]. To avoid unnecessary visual update and flickering while the developer focuses on code editing, we included a 500ms idle time for invoking updating. This architectural design makes it flexible to integrate the DemoScript Viewer with standalone IDEs such as Sublime Text[4] or Android Studio[5].

[1] https://nodejs.org/
[2] https://jquery.com/
[3] http://getbootstrap.com/
[4] http://www.sublimetext.com/
[5] https://developer.android.com/sdk/



```
/**
 * Task 3
 */
weave.launchMethod = weave.launchOption.default;

var service = function() {
  var action = '<button>next</button>';
  weave.select('.touchable[size="large"]')
   .startApp('Gmail')
   .on('tap', function(event) {
      event.getDevice().show(action);
   });
  weave.select('.shakable[size="small"]')
    .show('app launcher')
    .on('shake', function(event) {
      event.getDevice()
        .show('launching Maps...')
        .play('ding.mp3');
      weave.select('.showable')
       .startApp('GoogleMaps');
    });
};
```

```
09-25 12:33:31.629 Service loaded
09-25 12:33:31.629 Successfully deploy the service.
09-25 12:33:31.596 Editor is ready
```

Figure 6. The baseline condition in our evaluation. Similar to conventional IDEs, it provides a script editor and a set of emulators for testing.

EVALUATION
We hypothesized that DemoScript could help developers easily manage various aspects of the tasks for programming cross-device behaviors that would otherwise be challenging. To validate this hypothesis, we conducted a study to compare DemoScript with a baseline condition that provides a script editor and emulators—a typical setup for UI programming. Specifically, we used the Weave IDE that was previously introduced (see Figure 6) [7]. In the study, we asked participants to interpret the cross-device behaviors that are fulfilled a set of scripts, and then identify and fix issues in these scripts for achieving different behaviors. We garnered both quantitative measurement such as task completion time and qualitative feedback on their reactions to DemoScript.

Participants and Setups
We recruited 8 professional programmers (2 females), aged between 21 and 45 years (Mean=26) from an IT company. Participants were required to have moderate JavaScript programming knowledge and were selected randomly from volunteers via an internal study invitation. 5 out of 8 participants had limited mobile programming experience; only 1 had programmed wearable devices. None of them had used the Weave framework.

The study was conducted in a laboratory environment. A MacBook Pro running OS X and Google Chrome browser was connected to a 24-inch LCD display with 1920x1200 pixel resolution. We provided an external mouse and a keyboard. Each participant was compensated with a $40 gift card for their participation in a 90-minute session.

Procedures and Tasks
At the beginning of the session, we introduced the Weave framework to the participants and asked them to walk through a web tutorial by following a cross-device launch pad app design, which approximately took 10 minutes. The Weave API documentation was available to participants during the experiment.

We then asked the participants to perform two sets of tasks in sequence, denoted Set I and Set II, where Set I is assigned to the first condition and Set II for the second condition. We counterbalanced the order of the two conditions to guard against learning effects in our analysis. Each set consists of two tasks with varying complexity: the first task is relatively *easy* (denoted as Task 1) and the second one is *hard* (denoted as Task 2). Task 1 in each set has three subtasks, while Task 2 has two. We designed the tasks in the way that both sets have comparable complexity, shown as follows:

Task 1. Given a script with three device selections, two of them each had a callback for different event types, and the third selection was only initiated once inside a callback function. Not all the selections and actions matched the instructions. At most 15 minutes was given to finish the task.

Subtask 1. [*Script understanding*] Explain how a user would achieve the following goal: {set I – to see an image, set II – to launch the Maps app}. Describe what the user would see on each device.

Subtask 2. [*Script refinement*] Suppose the script's developer intended to show the {I – photo launch button, II – application} on a {I –wrist-worn, II – large-display} device, answer if the script works properly as described. If not, fix the script to match the app description.

Subtask 3. [*Interaction enhancement*] To improve the user experience, modify the script so that {I – a prompt is shown on the target device before the photo appears, and II – whenever there is any UI update, play a "ding" sound on the same device as an indication}.

Task 2. The provided script had the same number of selections as Task 1.

Subtask 1. [*Script understanding*] Given the app description, does the script work as described? What works and what is different? The app enables user to {I – shake her phone to launch Gmail with both auditory and visual feedback before switching the app; meanwhile, she would see the Gmail icon on a wearable device; II – shake a wearable device that has a thumbnail displayed in order to see a large view of the photo on another device with the Photos app; a "ding" sound is played before launching the Photos app}.

Subtask 2. [*Debugging*] Correct the script so that it matches the description.

Finally, participants were asked to complete an online questionnaire with 5-point Likert-scale questions and debrief their thoughts.

RESULTS AND DISCUSSION

We discuss participants' performances and experience in terms of both quantitative results and qualitative feedback on comparing the baseline system and DemoScript.

Task Performance

All the participants completed the given tasks in both conditions. The average completion time for each subtask is shown in Figure 7. Overall, using the baseline system, participants needed more time to fix and test a script (Mean=3.2 minutes compared to 2.8 minutes using DemoScript on average). Furthermore, participants had a higher error rate using the baseline method (17.5% compared to 2.25% with DemoScript). We recorded an error when the oral interpretation was incorrect or incomplete (for script understanding subtasks) or the script failed to fully achieve the instruction (for scripting tasks). We discuss these errors and participants' coding behaviors that we observed in the following section.

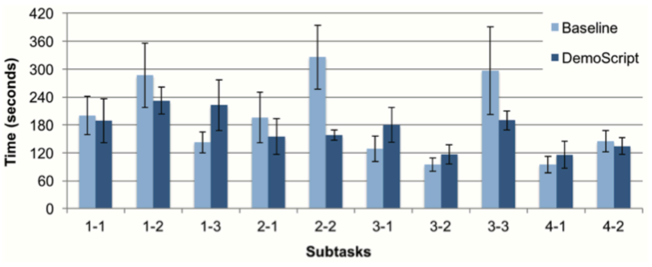
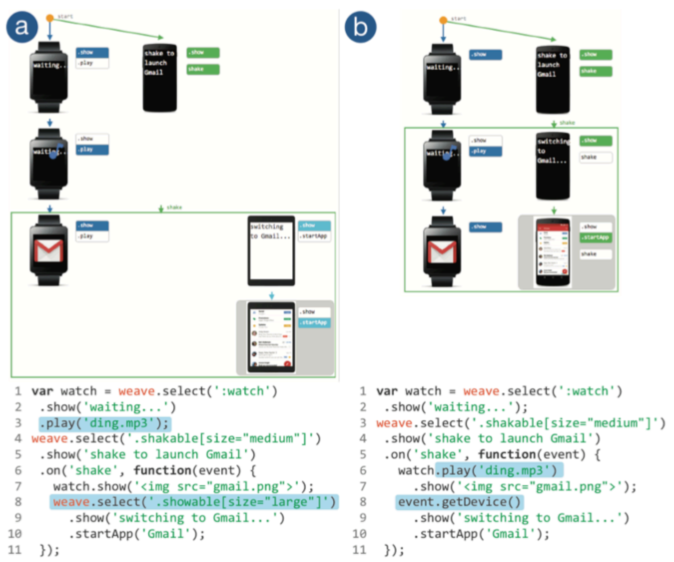


Figure 7. Average completion time for each subtask of the baseline system and DemoScript.



```
var watch = weave.select(':watch')
 .show('waiting...')
 .play('ding.mp3');
weave.select('.shakable[size="medium"]')
 .show('shake to launch Gmail')
 .on('shake', function(event) {
   watch.show('<img src="gmail.png">');
   weave.select('.showable[size="large"]')
     .show('switching to Gmail...')
     .startApp('Gmail');
 });
```

```
var watch = weave.select(':watch')
 .show('waiting...');
weave.select('.shakable[size="medium"]')
 .show('shake to launch Gmail')
 .on('shake', function(event) {
   watch.play('ding.mp3')
     .show('<img src="gmail.png">');
   event.getDevice()
     .show('switching to Gmail...')
     .startApp('Gmail');
 });
```

Figure 8. An example showing how developers can identify and correct errors with the support of storyboard.

Coding Behavior

Identifying Design Patterns and Redundancy

DemoScript's storyboard visualization helped participants identify errors where the provided script did not match the provided application description. For example, in the original script of task 3 (see Figure 8a), auditory feedback was played before the event is triggered, and the script incorrectly specified a wrong device selection to initiate the application. These errors were visually distinguishable in the storyboard, and all the 4 participants were able to fix the script with DemoScript (see Figure 8b), whereas 2 of the other 4 participants failed to correct all the errors using the baseline system.

The storyboard also helped participants to quickly identify the redundant code. Only 2 of 4 participants using the baseline correctly removed the redundant code in Task 3. Participants explained, the advantages of DemoScript include "*Being able to see the various device actions in sequence*" (P3) and to see "visualization of subsequent events" (P6) or a "*full overview over the execution flow; event listeners are clearly visible*" (P7).

Encouraging Testing When Scripting

Using DemoScript, participants actively navigated in the Editor to test the logic in the given scripts line-by-line and to verify their code changes (25.35 line navigations on average). They found the concept of the multi-device storyboard straightforward (4.5, SD=0.76) and it was easy to test (4.625, SD=0.52). On the contrary, participants only tested the code 2.23 times on average using the baseline system, which provided a "Run" button to deploy the code to emulators. P5 explained, with DemoScript, "*You can clearly see your code running on the screen, which makes it easier to debug and understand what you are doing instead of trying to play the scenes inside your head. It was very clean and well designed interface and it was easy to interact with it.*" Compared to standard IDEs for mobile devices, P2 shared, "*It was a big advantage to have an emulator for each device class running simultaneously and it was nice that the emulators ran the program instantly.*"

In the baseline condition after using the DemoScript system, two participants answered and scripted without testing the scripts. One of them asked if he could pull up the device list as what DemoScript showed. The participant later explained that he preferred to focus on coding, so DemoScript made it easy to see the runtime results while he scripted, whereas using the baseline system, he thought through the code and was confident in his interpretation or fixes (note that 2 of 5 subtasks failed).

We suspect that the ability of line-by-line testing improved the task success rate as it helped developers identify errors early. Participants constantly verified the script visually when coding, which was especially effective when learning a new framework. When participants found an error, they could quickly identify, reason, and correct the code. This might also lead to the fact that for some subtasks (1-3, 3-1, 3-2, 4-1), participants spent more time using DemoScript.

Visualizing Device Examples

Participants preferred how DemoScript presented a list of emulators and showed the mapping between selectors and emulators (4.625, SD=0.52): "*With the initial IDE, it was more difficult to find out which devices were being selected*" (P1). For the subtasks of fixing the selector, 4 of 8 participants chose to create a selector from device examples. P2 noted that "*The other nice thing was the device selector string generator.*"

Participants found it easy to learn (4.5, SD=0.53), script cross-device interactions (4.75, SD=0.46), and felt capable of scripting the provided tasks (4.875, SD=0.35) with DemoScript. Participants also expressed the interests in using DemoScript if it becomes available (4.75, SD=0.46): "*It is awesome and I want to use it :D cant wait for it.*" (P5)

Opportunities and Limitations

We also collected user feedback on the additional supports that they expected DemoScript could provide. First, handling more complicated logic, such as conditioning when a callback function breaks into two different behaviors based on certain condition using `if-else` and `switch`. Our revised design providing basic support of language-based condition is to arbitrarily select one condition and render the storyboard but provide other paths for developers to test. Second, two participants suggested combing the baseline and DemoScript IDEs—to provide a new tool that can interactively shows the step-by-step execution but also allow them to do formal testing with emulators. Debuggers of popular IDEs enable developers to test application logic by inspecting execution states (e.g., checking variable values at breakpoints). In contrast, DemoScript provides high-level visualization of application logic and execution, which allows developers to easily grasp important aspects of interaction flows. These two approaches complement rather than compete with each other. A developer might find it beneficial to switch between the unique advantages of both. Third, the storyboard can be possibly generated as a visual instruction for consumers who would interact with the final application at runtime.

Beyond supporting cross-device programming based on the Weave framework, we argue that many components of DemoScript could apply to other types of app development. Techniques for selecting interactive elements (devices in our case) and providing UI feedback and listening to user events (such as touch or sensor input) are common in interaction or mobile frameworks such as Android. To apply our techniques, one would need to analyze UI constructs in the framework, which is conceptually straightforward. However, it would require additional engineering effort to analyze a program at various levels.

CONCLUSION

We presented DemoScript, a technique that automatically analyzes and visualizes a cross-device interaction program while it is being written. Particularly, we introduced cross-device storyboards, a novel visualization for cross-device development. It is closely coupled with scripting by offering step-by-step execution of a selected portion or the entire program. Via the storyboard, a developer can revise various aspects of a program by direct manipulation. We evaluated DemoScript with 8 professional programmers and found that it outperformed the baseline condition in many ways to simplify programming tasks for complex interaction behaviors.

REFERENCES

1. Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. 2012. DataPlay: interactive tweaking and example-driven correction of graphical database queries. In *Proc. UIST '12*, 207-218.
2. Apple Inc. Swift. 2015. https://developer.apple.com/swift/
3. Apple Inc. WatchKit. 2015. https://developer.apple.com/watchkit/
4. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proc. ICSM '98*, 368-377.
5. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12, 2301-2309.
6. Xiang 'Anthony' Chen, Tovi Grossman, Daniel J. Wigdor, and George Fitzmaurice. 2014. Duet: exploring joint interactions on a smart phone and a smart watch. In *Proc. CHI '14*, 159-168.
7. Pei-Yu (Peggy) Chi and Yang Li. 2015. Weave: Scripting Cross-Device Wearable Interaction. In *Proc. CHI '15*, 3923-3932.
8. Stephan Diehl. 2007. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
9. James Fogarty, Desney Tan, Ashish Kapoor, and Simon Winder. 2008. CueFlik: interactive concept learning in image search. In *Proc. CHI '08*, 29-38.
10. Jens Grubert, Matthias Heinisch, Aaron Quigley, and Dieter Schmalstieg. 2015. MultiFi: Multi Fidelity Interaction with Displays On and Around the Body. In *Proc. CHI '15*, 3933-3942.
11. Google Inc. Android Wear. 2015. https://www.android.com/wear/
12. Google Inc. Chrome Apps. 2015. https://developer.chrome.com/apps/
13. Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective physical prototyping through integrated design, test, and analysis. In *Proc. UIST '06*, 299-308.
14. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proc. UIST '08*, 91-100.
15. Ariya Hidayat. Esprima: ECMAScript parsing infrastructure for multipurpose analysis. 2015. http://esprima.org/
16. Steven Houben and Nicolai Marquardt. 2015. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proc. CHI '15*, 1247-1256.
17. Melody Y. Ivory and Marti A Hearst. 2001. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv*. 33, 4, 470-516.
18. Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. 2011. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proc. UIST '11*, 217-224.
19. Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: integrated support for developing interactive camera-based programs. In *Proc. UIST '12*, 189-196.
20. Scott R. Klemmer, Mark W. Newman, Ryan Farrell, Mark Bilezikjian, and James A. Landay. 2001. The designers' outpost: a tangible interface for collaborative web site. In *Proc. UIST '01*, 1-10.
21. James A. Landay and Brad A. Myers. 1996. Sketching storyboards to illustrate interface behaviors. In *Proc. CHI '09*, 193-194.
22. Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R. Klemmer. 2010. Designing with interactive example galleries. In *Proc. CHI '10*, 2257-2266.
23. Yang Li and James A. Landay. 2008. Activity-based prototyping of ubicomp applications for long-lived, everyday human activities. In *Proc. CHI '08*, 1303-1312.
24. Yang Li, Xiang Cao, Katherine Everitt, Morgan Dixon, and James A. Landay. 2010. FrameWire: a tool for automatically extracting interaction logic from paper prototyping tests. In *Proc. CHI '10*, 503-512.
25. Henry Lieberman and Christopher Fry. 1995. Bridging the gulf between code and behavior in programming. In *Proc. CHI '95*, 480-486.
26. James Lin and James A. Landay. 2008. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In *Proc. CHI '08*, 1313-1322.
27. Microsoft Inc. HoloLens. 2015. https://www.microsoft.com/microsoft-hololens/
28. Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact*. 7, 1, 3-28.
29. Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. 2014. Interactive development of cross-device user interfaces. In *Proc. CHI '14*, 2793-2802.
30. Donald A. Norman and Stephen W. Draper. 1986. *User Centered System Design; New Perspectives on Human-*

Computer Interaction. L. Erlbaum Assoc. Inc., Hillsdale, NJ, USA.

31. Stephen Oney and Joel Brandt. 2012. Codelets: linking interactive documentation and example code in the editor. In *Proc. CHI '12*, 2697-2706.
32. Roman Rädle, Hans-Christian Jetter, Mario Schreiner, Zhihao Lu, Harald Reiterer, and Yvonne Rogers. 2015. Spatially-aware or Spatially-agnostic?: Elicitation and Evaluation of User-Defined Cross-Device Interactions. In *Proc. CHI '15*, 3913-3922.
33. Daniel Ritchie, Ankita Arvind Kejriwal, and Scott R. Klemmer. 2011. d.tour: style-based exploration of design example galleries. In *Proc. UIST '11*, 165-174.
34. Sifteo Inc. Sifteo Cubes. 2015. https://www.sifteo.com/
35. Michael Toomim, Andrew Begel, and Susan L. Graham. 2004. Managing Duplicated Code with Linked Editing. In *Proc. VLHCC '04*: the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, 173-180.
36. Khai N. Truong, Gillian R. Hayes, and Gregory D. Abowd. 2006. Storyboarding: an empirical determination of best practices and effective guidelines. In *Proc. DIS '06*, 12-21.
37. Jishuo Yang and Daniel Wigdor. 2014. Panelrama: enabling easy specification of cross-device web applications. In *Proc. CHI '14*, 2783-2792.
38. Mark Weiser. 1981. Program slicing. In *Proc. ICSE '81*: the 5th International Conference on Software engineering, 439-449.