UI Testing Cross-Device Applications

Maria Husmann, Michael Spiegel, Alfonso Murolo, and Moira C. Norrie
Department of Computer Science, ETH Zurich
{husmann, murolo, norrie}@inf.ethz.ch, michael.spiegel@alumni.ethz.ch

ABSTRACT

The increasing number of devices available to a user has prompted the research community to explore how these can be used in combination. Frameworks and toolkits have been proposed to facilitate the design and implementation of these cross-device applications. Still, implementing cross-device applications remains complex because of the fragmentation of the user interface and logic across devices and the flexibility required to adapt to different combinations of devices. Testing in particular has been identified as a critical challenge. To address these issues, we introduce XD-Testing, a library that provides explicit and implicit device selectors, device templates and scenarios, as well as a visualiser for application screenshots. In a case study, we demonstrate how we used the library to author human-readable tests for a cross-device gallery that verify if a UI distributes correctly and if it works as expected *despite* being distributed.

Author Keywords

multi-device, distributed user interfaces; cross-device; ui testing; automated testing;

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation : User Interfaces - Graphical user interfaces

INTRODUCTION

The number of computing devices available to a user has increased in recent years and now includes smartphones, smartwatches, smartTVs, tablets and even public screens. This move away from a single personal computer to an ecology of devices has prompted researchers to study how people currently use these in combination and how this combined use could be improved further [24]. To this end, a number of cross-device interaction techniques have been proposed (such as [19, 12]). Furthermore, applications that make use of and distribute their user interface (UI) across multiple devices have been implemented and studied. Proposed scenarios cover both single- and multi-user cases and range from travel planning [13, 7], (collaborative) sensemaking [10, 29], and interaction with geospatial data [27] to distributed video players and photo sharing [25].

ISS'16, November 06–09, 2016, Niagara Falls, ON, Canada

ACM 978-1-4503-4248-3/16/11…$15.00
DOI: **http://dx.doi.org/10.1145/2992154.2992177**

Some multi-device systems have been built for a specific, ideal set of devices (such as a combination of tabletops and tablets), whereas others aim to adapt to the set of devices at hand [20]. While the first approach is better suited to relatively controlled environments such as smart rooms, the second one is better for supporting adhoc interaction where the set of available devices might not be known in advance. For example, a system designed to include a tabletop might be hard to use when there is not one available, while a more flexible system could assign the role that the tabletop had to another device such as a tablet. Zagermann et al. have shown in [31] that this can be done without incurring a performance penalty in a sensemaking task.

Building these flexible cross-device applications is complex. While the design, prototyping, and implementation stages are supported by various cross-device frameworks and toolkits, in contrast, testing remains largely unaddressed. However, it has been identified recently as a critical challenge in the development of multi-device experiences [6]. Existing testing tools have not been built with cross-device applications in mind and offer limited support for their inherent challenges. These challenges include the large space of possible device combinations that could be used, in particular in multi-user scenarios. Testing all of them manually and repeatedly would be tedious. However, omitting tests for the sake of convenience could lead to regressions in later iterations. Another obstacle to testing is the fragmentation of the application across multiple devices. An interaction on one device can have an effect on another device and, given the flexibility of the application, it can be hard to predict in advance which device that will be as it could depend on the set of devices currently in use.

Given that testing is an integral part of established software engineering practice, we consider this lack of support of testing an obstacle to the wider dissemination of cross-device applications. To tackle this obstacle, we have created XD-Testing, a library for automated testing of web-based cross-device applications that makes the following contributions.

- A domain specific language for addressing devices in cross-device settings implicitly and explicitly, thus reducing the impact of the fragmentation on writing tests.
- Support for configuration and execution of tests with different sets of devices, which can be either crafted by the tester or generated by our system.
- Recording screenshots and visualising the application at critical checkpoints defined by the tester, allowing the comparison of behaviour on different sets of devices.

The rest of the paper is structured as follows. We first review related work on cross-device development and testing. We then introduce a motivating example to discuss the challenges particular to cross-device testing. Next we introduce our testing library and give a brief description of its architecture and implementation. Then, we report on the evaluation of the library in a case study with a cross-device application. We conclude with a discussion of the limitations and directions for future work.

BACKGROUND

Our research builds on two areas of related work: cross-device development and software testing. To understand the challenges associated with testing cross-device applications, we first introduce how these are built by discussing relevant cross-device frameworks and existing tools. Then we provide background on software testing with a focus on testing user interfaces.

Cross-Device Frameworks and Tools

Applications that distribute their interfaces across multiple devices have a long tradition in ubiquitous computing research (for example [28]). The more recent proliferation of internet-connected consumer devices such as tablets and smartphones has renewed interest in the topic (for example [9]). At the same time, increasingly powerful web technologies (such as WebSockets) have made the web a promising platform for building cross-device applications, eliminating the need to write separate code for each operating system and providing a lightweight delivery mechanism via URLs.

A new generation of frameworks and toolkits [16, 1, 25] has tapped into the potential of the web and provides abstractions for building cross-device applications. These frameworks typically provide concepts for connecting devices to form a group, specifying how the user interface should be distributed across them and keeping the application state synchronised among the devices. Some [23, 18, 4, 26] also abstract away sensors and provide higher-level proxemic and context information. For specifying the UI distribution, different approaches can be observed. Connichiwa [25] and Weave [4] provide an *event-based* mechanism to imperatively show and hide content on devices upon connection or disconnection of other devices. In contrast, Panelrama [30] allows developers to declare *affinity scores* for parts of the UI (panels) which the system uses to automatically assign that panel to a suitable device. For example, a video panel would be distributed to a device with a large screen, while a control panel would be distributed to one that is ready to hand. A third approach is to distribute the UI based on a more general model of *context rules* [8]. A context rule consists of a trigger and an action that will be executed upon the trigger. Proposed trigger dimensions are the device, the user, and the environment as well as social information, whereas actions include showing and hiding UI elements on specified devices.

Despite these mechanisms, it can be challenging for a developer to predict how the UI will be distributed given a set of devices. With *event-based* approaches, the distribution may depend on past interactions and the order that events (such as devices connecting) are received. To alleviate this issue, Weave was complemented with DemoScript [5] which generates previews of the step-by-step execution of the program. However, it assumes a given set of devices and cannot contrast how program execution may differ depending on the set of available devices. Using Panelrama, developers struggled with choosing suitable *affinity scores* and this prompted the authors to build a preview tool. Finally, a large set of *context rules* could make it challenging to understand which rules would be applied, possibly at the same time, and what the combined application of multiple rules would produce.

Given these insecurities, we consider it crucial that developers can easily test their applications with different sets of devices and verify that the distribution is acceptable in each case. This has been addressed in part by tools that emulate devices and thus allow developers to work with devices that are not physically present. XDStudio [21] supports the developer in specifying distribution profiles and checking them on emulated devices. However, the focus is on the design phase and there is no easy way to repeatedly test different sets of devices. Similarly, WatchConnect can emulate smartwatches for prototyping cross-device interactions, but also has no support for automated testing. In contrast, XDSession [22] explicitly addresses testing and debugging. It supports the emulation of one device at a time and allows recorded interactions to be replayed. Switching between different sets of devices is thus labour intensive and no automation is provided beyond the replay mechanism. The first issue has been addressed by XDTools [14] that allows sets of devices to be named, stored, and reloaded. However, XDTools focuses on manual testing and debugging rather than providing a mechanism for specifying formal tests and automating their execution.

Testing

Software testing can be executed at different levels of granularity and with different levels of coupling with the software being tested. On one end of the granularity spectrum are unit tests that verify isolated program behaviour at the smallest level of the system decomposition (such as single functions and classes) [2]. At the other end, there is system testing. A specific part of the latter, namely UI testing, also referred to as end-to-end testing, focuses on a whole, integrated system mimicking real-world interactions with the user interface.

UI testing also has less coupling with the underlying code, realising a black-box testing approach which focuses on the expected input/output behaviour of the system. Conversely, white-box testing approaches focus on the internal structure of the components, aiming at testing every possible state and interaction between the system components [2].

Unit tests do not suffer much from the fragmentation and distribution in cross-device applications, as they operate in isolation and at a high level of granularity and the behaviour under test would typically be executed on a single device. However, UI testing is highly impacted by the fact the application is distributed across multiple devices which need to be coordinated during testing. For example, input on one device can trigger output on another device. Thus in this paper, we focus our research on UI testing.

For UI testing, two main approaches can be observed [17]. Interactions with the UI can be recorded and replayed (for example as in Selenium[1] for browsers) or interactions can be scripted by the tester and executed by an automation tool. While the second approach requires more time from the developer to carefully craft the interactions, it is easier to adapt the scripts when the software changes [3], while recordings need to be re-captured if the changes are significant. A previous study [17] on web applications found that the scripted approach can cumulatively become less expensive than the record and replay already after one to three releases. These results show that the higher costs of scripting the test cases for UI testing can be quickly amortised with reduced costs of test maintenance in subsequent releases.

Cross-device applications typically aim to support a diverse set of devices, including tablets or phones, while they are often developed on desktop or notebook computers with larger screens. The challenges of testing mobile devices have been addressed by Testdroid [15], which allows testers to record scripts and replay them on multiple devices in parallel or alternatively on an emulated mobile device. Hesenius et al. [11] also provide support for testing on mobile devices with an extension to the test automation framework Calabash[2] for gesture-based interaction. However, coordination among the devices is considered in neither work and hence they are not adequate for cross-device testing.

MOTIVATING EXAMPLE

Two properties of cross-device applications introduce challenges into the testing process when compared to single-device applications: First, the fragmentation of the application across multiple devices, and second, the flexibility of the system to adapt to different device sets.

To illustrate these challenges we will use the example of a cross-device video player, similar to the one used in Panelrama [30]. The player consists of the actual video, a set of controls, and a search interface that lists videos matching a query. An example test for such a player could be to ensure that, when a video is selected from the list of search results, it will be displayed.

```
1 client
2   .click('li=My Video')
3   .getText('#title')
4   .then(function(value) {
5     assert(value === 'My Video');
6     // true
7   });
```

Listing 1: A simple test case that checks if a user clicks on a video in a list that the title of the video is displayed.

Listing 1 gives a simplified version of how such a test could be written. First, a click on the list element is triggered, then an assertion checks if the title element associated with the video shows the correct text. The example test will work fine, if both the search results and the video are on the same device. However, it cannot handle the case where these two elements are distributed across multiple devices. As a next step, the tester could script a test that contains two devices and explicitly test that particular case. They would need to pair the devices first, then the list item should be clicked on the device that shows the search. Identifying that device may not be trivial, as it can depend on properties of the device itself (such as its screen size), the other devices, or the context in general. Finally the tester needs to ensure that the title element is only checked after the changes have been propagated to the second device. In addition to the increased effort compared to writing the single-device test case, this would only test one particular combination of devices. However, selecting a video from the search results should *always* update the title, independent of how the UI is distributed and the number and type of devices involved. Existing testing libraries offer no constructs tailored towards expressing such tests that *implicitly* assume a distributed user interface and that could be written once and then be executed parametrised with different sets of devices.

While abstracting away the distribution of the UI could facilitate writing tests such as the one outlined above, addressing it *explicitly* would allow the developer to verify that the UI is distributed as expected. While cross-device applications often aim at supporting a diverse set of devices, they could still be tailored to support some combinations especially well. In the case of the video player, these could include {phone, TV}, {phone, tablet, TV}, {phone, phone, phone, TV}, and {phone, tablet}. An application could be tested with these combinations explicitly, similar to the way responsive web applications are typically tested on phones, tablets, and desktop PCs. As discussed in the previous section, it can be challenging for a developer to predict how the UI would be distributed across a given set of devices. Tweaking an affinity score or adding a context rule that attempts to cover one set of devices could influence the distribution of the UI on another set. Having this covered by an automated test spares the developer from having to test these distributions manually and can avoid regressions. A test for a cross-device application could also specify properties of a distribution that should always hold. For example, if a phone is among the devices, it should show the controls, and similarly, a TV should show the video. Also, each important part of the UI (the video, the search, and the controls in our example) should always be allocated to at least one device. Crafting tests of this kind with existing tools requires a lot of effort from the developer. They need to manually create a representation of each device in their test, connect all the devices to each other, ensure that state synchronisations are awaited and keep track of all devices in each individual test.

Finally, even if a tester carefully crafts tests for some combinations of devices, they would only test the device combinations they expected. While these are certainly important to test, it can be challenging for a developer to predict what devices might be used to access an application. The space of potential device combinations is huge, and continually evolving,

[1] http://www.seleniumhq.org/
[2] http://calaba.sh/

Template property	Supported values
name	Strings
type	*"watch"*, *"phone"*, *"tablet"*, *"desktop"*
size	*"xsmall"*, *"small"*, *"medium"*, *"large"*
width	Positive integer
height	Positive integer
desiredCapabilities	Accepted by Selenium

Table 1: Properties and supported values of device templates.

so trying to cover any possible combination is challenging if not impossible.

XD-TESTING

To address the challenges introduced by the fragmentation and flexibility of cross-device applications, we introduce XD-Testing, a testing library tailored towards web-based cross-device applications. XD-Testing consists of three main components: a JavaScript API for defining and randomly generating device scenarios for parametrised testing, a DSL for explicit and implicit testing of distributed UIs, and a mechanism for recording and visualising program flow in cross-device applications. XD-Testing extends WebDriverIO[3] which provides programmatic commands to remote control browsers with Selenium.

Device Templates and Scenarios

We offer a mechanism to parametrise a cross-device test with a set of devices so that the same test can be executed with different sets of devices. To this end we introduce *device templates* and *device scenarios*. A device template represents a reusable description of a device. We define a device scenario as a set of devices that interact with each other. Device scenarios can be built by instantiating devices from templates.

```
1 {
2     name: "Nexus 5",
3     type: "phone",
4     size: "small",
5     width: 360,
6     height: 640,
7     desiredCapabilities: {browserName:
          "chrome"}
8 }
```

Listing 2: An example device template for a Nexus 5 phone.

Table 1 lists all supported properties and values of a template and Lst. 2 shows an example. Each device template contains a human-readable name, the device type, the device screen dimensions with the assigned screen size and the desired capabilities as required by Selenium. *Width* and *height* need to be specified in density-independent pixels (dp), a measurement that reflects the physical size of a device rather than the pixel density. XD-Testing comes with a set of 8 predefined templates, more can be added by the tester.

A tester can either craft a scenario directly in JavaScript (Lst. 3) or load it from a configuration file. The second approach supports easier reuse of scenarios across multiple tests and even across different projects. XD-Testing includes a command line tool for generating device scenarios. The aim of the tool is to provide the tester with a wide range of (possibly unexpected) combinations of devices but also to lessen the workload required for creating the scenarios. The tester can constrict the randomness to some extent by limiting the maximum number of devices per scenario and restricting their properties (as defined in the templates) to a given subset (for example to *small* and *medium* devices). Furthermore, the output is human-readable and can be edited by the tester.

```
1  var scenario = {
2    ctr: xdt.templates.devices.nexus4(),
3    dA: xdt.templates.large.nexus10(),
4    dB: xdt.templates.desktops.chrome()
5  };
6
7  xdTesting.multiremote(scenario).init()
8      // Load url on all devices
9      .url("http://myapp.com")
10     .click("button");
```

Listing 3: Creating a device scenario ad-hoc through JavaScript: Templates can be accessed from a general set (line 2) or by characterisation such as size (line 3) or type (line 4). A test is parametrised with the scenario (line 7) so that all following steps (line 9 and 10) are executed on the chosen devices.

Domain Specific Language

XD-Testing features a domain specific language providing a clear and concise syntax for testing cross-device applications. The DSL supports method chaining which means that each method returns an object that again can receive a method call. This allows several methods to be chained together and results in shorter code. Furthermore, the DSL abstracts away asynchronism, thus making it easier to read and write tests.

The main goal of our DSL is to facilitate the coordination of multiple devices when authoring tests. While our foundation, WebdriverIO, includes basic functionality to simultaneously control multiple browsers, it is restricted to either broadcast commands to all browsers or pick a single browser by its identifier. With XD-Testing, we introduce novel device selectors allowing a developer to choose a device subset explicitly or implicitly based on criteria such as device identifier, type, display size and accessible application elements, while maintaining readability and syntax consistency. Matching devices are passed to a callback function which executes clearly separated from the initial device set. Commands within the callback context are only executed on the selected device subset. The commands following the selection address the initial device set and are executed after the previous actions on the device selection are finished, allowing a test to leave a selected device subset and continue on the original set. Device

[3] **http://webdriver.io/**

selections can also be nested and support step-wise narrowing of the device specifications by combining several criteria, for example by first selecting devices by element and then restricting the resulting set to tablets only.

Explicit Device Selection

Explicit device selection is based on static device characteristics as defined in the device templates and on dynamic properties evaluated at run-time. This type of selection is used to explicitly choose a set of devices and address them with commands. The library provides several selector commands with different selection criteria. Supported criteria are the characteristics defined in the device scenario, such as device id, type and size, and the element selector (Table 2). The element selector matches all devices where a given HTML element is visible. These device selectors create a set of devices that consists of zero or more matching devices. Furthermore there is the `selectAny` command which selects a single arbitrary device or throws an error when called on an empty set. This is useful to ensure that only a single device is chosen. Nesting the *any* selection into another selection allows the tester to choose a single device from a set of devices which already matches certain criteria. Listing 4 illustrates this use case: first a button is clicked on all tablet devices, then it is clicked on just a single tablet device.

Command	Criteria Category	Criterion
selectById	Device characteristic	One or multiple device Ids
selectBySize		One or multiple screen sizes
selectByType		One or multiple device types
selectByElement	Dynamic	A element selector
selectAny		None, arbitrarily chosen

Table 2: Explicit device selection commands.

```
1 .selectByType("tablet", tablets =>
     tablets
2   .click("button")
3   .selectAny(device => device
4     .click("button")
5   )
6 );
```

Listing 4: First, all tablets are selected with a *type* selector (line 1), then the selection is constrained to a single device with the *any* selector (line 3).

Explicit device selection commands accept a second callback to address the *complementary device set*. This allows the original device set to be split into two subsets using the specified criterion. The first callback provides the devices matching the criterion (for example all tablets), while the second callback is applied to the complementary device set (the original device set minus all tablets). The complementary callback can be useful in combination with the `selectAny` command. In this case, the second callback returns all devices except the one returned in the first callback. This allows a developer to execute an action on one device and check results on all other devices. Listing 5 provides an example of use of the complementary callback with our video player example. If multiple devices show the controls, the state of the play/-pause button should be consistent across all of them. This can be tested by selecting all devices that have controls (line 1) and then picking a single one (line 2 and 3) and clicking the play button (line 4) on the selected device. The complimentary callback (lines 5 to 7) waits for all other devices that have controls to update their play button to display the text *Pause* and the test will fail if any device does not comply within the given timeout.

```
1 .selectByElement("#controls",
     controller => controller
2   .selectAny(
3     one => one
4       .click("#play"),
5     rest => rest
6       .waitForVisible("#play=Pause",
           2000)
7   )
8 );
```

Listing 5: Example usage of the complementary device set.

Implicit Device Selection

Implicit device selection handles cross-device behaviour of the application in an abstract way. It considers the distributed UI in its entirety by removing the need to address specific devices and instead it dynamically chooses the appropriate devices for each command. The test developer does not need to specify on which device the command should be executed. Instead, the system analyses the command and finds devices that can execute it. For example, if a command states that a specific UI element should be clicked, the system will find the devices that show the specified element and execute the command only on these devices. The semantics for implicit device selection are defined as follows:

- `implicitAll`: The system ensures that each command is executed on *at least one* device. Each command is guaranteed to be executed on all devices that match the selector given in a command.

- `implicitAny`: The system ensures that each command is executed on *exactly one* device. While `selectAny` chooses a device and then tries to execute all commands on the same device, `implicitAny` chooses a suitable device for each command.

In both case, the library throws an error, if no device is matched. Using our motivating example, we outline the differences in the semantics. Listing 6 uses the `implicitAny` and Lst. 7 the `implicitAll` selector. The `implicitAny` example first looks for a device that has the list item with the text *My Video*. It will click that item on exactly one device (or throw an error if there is no such device). Then it will select any single device that has the title element and check that the text matches *My Video*. The main difference of the `implicitAll` example compared to the `implicitAny`

example is that each command will be executed on all devices that match the selector in the command, potentially resulting in multiple clicks. As each command could be executed on multiple devices, we cannot assume a single return value for commands that provide a result (such as `getText`). In our example, multiple devices could show the title element and all of them need to be checked. This is is done in lines 5 to 6 of Lst. 7 by accessing the `arguments` property of the function. The two selectors could also be combined to first click the list item on exactly one device and then verify that the title is correct on all devices that have the title element.

```
1 .implicitAny(device => device
2   .click('li=My Video')
3   .getText('#title')
4   .then(function(value) {
5     assert(value === 'My Video');
6   }));
```

Listing 6: A simple test case using the `implicitAny` selector. Each command is executed on exactly one device.

```
1 .implicitAll(device => device
2   .click('li=My Video')
3   .getText('#title')
4   .then(function() {
5     var values = Array.prototype.slice
          .call(arguments);
6     values.forEach(value => assert(
          value === 'My Video'))
7   }));
```

Listing 7: A simple test case using the `implicitAll` selector. Each command is executed on one or more devices.

Device Set Commands

The XD-Testing library provides additional commands that provide control of the devices:

- `getCount` returns the number of devices contained in the device set.
- `forEach` provides access to each individual device in a set and accepts a callback with two parameters. The first parameter provides an individual device and allows the tester to execute commands on it. The second parameter gives a consecutive index for each device, i.e. the first device is identified by 0, the second by 1, etcetera.

Flow Recording and Visualisation

In addition to the formal testing that we address with the concepts introduced above, we also support more informal testing. We provide a visual representation of an application at interesting points in the execution. This allows a tester to visually verify the application behaviour. To this end, XD-Testing records the execution of each command on each device with a screenshot. We define a *flow* as the compound assembly containing information about the participating devices, the executed commands, and recorded screenshots.

Recording

Each library execution generates a flow and stores its JSON representation into a `.json` file. A flow file contains all necessary information about an execution and can be shared with other developers. The following commands are provided for flow recording.

- `name(name)`
- `checkpoint(name)`
- `addErrorCheckpoint()`

The `name` command is used to define a unique and human-readable name for each flow. It is used in the visualiser to identify flows. Developers can mark interesting points in the execution by defining a named checkpoint using the `checkpoint` command. While the checkpoint names for a single device need to be unique, different devices can use the same checkpoint name. Checkpoints with identical names will be grouped together across multiple devices in the visualisers. `addErrorCheckpoint` marks the flow as failed by adding an "ERROR" checkpoint with the remaining commands and screenshots. In addition to marking failures individually, this also allows a developer to add error checkpoints globally by creating a test hook that checks after each test if a failure has occurred (Lst. 8).

```
1 afterEach(function() {
2   if (this.currentTest.state ==
        'failed') {
3     return devices
4       .addErrorCheckpoint()
5   }
6 });
```

Listing 8: Adding error checkpoints globally.

Visualisation

XD-Testing provides a basic visualisation of recorded flows and allows the comparison of multiple flows side by side. The visualiser presents a list of all flows that are found in a specified directory. Flows that contain error checkpoints are marked visually. For a chosen flow, all checkpoints are listed and can be used to navigate through the flow. Optionally, all executed commands can be shown. Multiple flows can be displayed next to each other (Fig. 1), allowing the tester to visually compare flows. This could be useful for comparing the same interaction with different device scenarios or to identify visual regressions by comparing the same interaction with different versions of the code.

ARCHITECTURE AND IMPLEMENTATION

Since XD-Testing focuses on UI testing, its architecture has been conceived with the goal of controlling multiple browser instances at the same time, all of them running the client-side portion of the software under test (SUT). Figure 2 presents an overview of the architecture of the entire system.

XD-Testing is deployed as an NPM package running in Node.js[4] and it can execute the test case in a device scenario,

[4] **https://nodejs.org/en/**

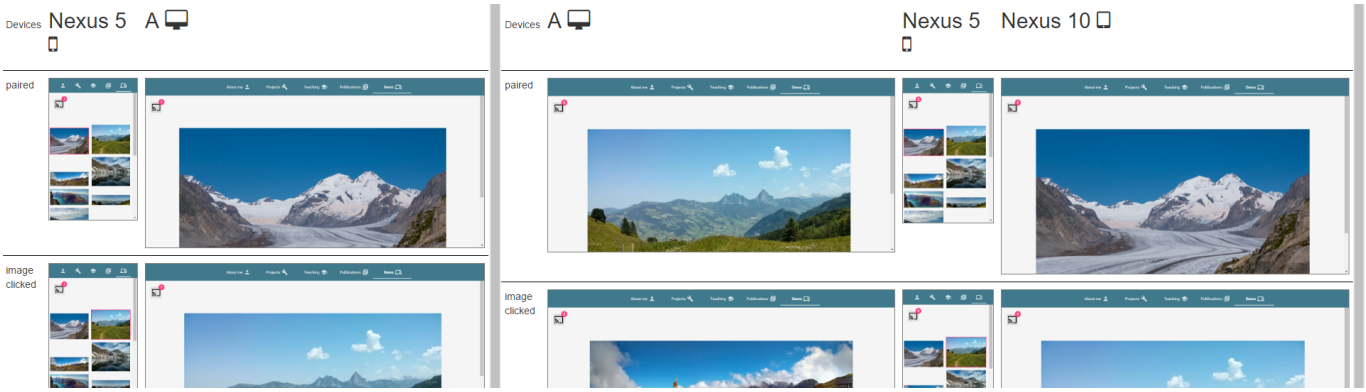

Figure 1: Two flows being compared with the visualiser: The one on the left has been generated with a scenario consisting of a phone and a larger screen, the one on the right additionally contains a tablet.

Figure 2: Architecture of XD-Testing. On the left side, the Node.js server required to run XD-Testing; on the middle/right side, the integration with Selenium Server, its counterpart executing UI Testing through browser control.

coordinating the test-case steps across the different browsers. It is built on top of WebDriverIO, which can control any number of browser instances in a Selenium server by sending commands that follow the WebDriver protocol[5].

The Selenium server is able to process commands received through WebDriver to execute the corresponding operations on the browser instances. The connection between Selenium and each browser requires a separate driver for each browser type. For example, Chromedriver[6] is a driver that implements the WebDriver protocol, and is capable of controlling instances of Chrome. XD-Testing supports any browser in the device scenarios, as long as it can be managed through a Selenium server.

Browsers are the clients of a cross-device web application under test. They will be instantiated depending on the device scenario that has been specified for the test run; in addition, depending on the application and the device scenario, each browser may show similar or different user interfaces on which the test will be run.

XD-Testing extends the functionalities of WebDriverIO by adding the cross-device abstractions and introducing some additional tracking behaviour, which is required in implicit contexts and in screenshot generation. As an extension, XD-Testing can act as a replacement for WebDriverIO and is backwards compatible with it. This can be useful in a scenario in which a legacy application without cross-device features (which was tested with WebDriverIO) is extended with cross-device features. In this case, the developer only needs to replace the WebDriverIO object in their legacy test cases with the XD-Testing object, and then write new test cases that aim at supporting the new cross-device functionality.

The chosen architecture easily allows the integration of XD-Testing in more extensive and automated testing workflows, such as frameworks for test automation that can help systematically define test suites to be run repeatedly, for example for regression testing. In our case, we used Mocha[7] to define test suites written with XD-Testing, even those to validate XD-Testing itself. In addition to the definition of test suites, Mocha allows developers to attach hooks before and after individual test cases (or test suites) are executed. This can be used in combination with some useful interface abstractions that XD-Testing provides for the integration with test runners, such as a function to generate a checkpoint in the event that a test case in the runner has failed (see Lst. 8). Further interface abstractions allow the URL of the application under test to be globally set, or the length of timeouts to be waited when asserting effects of wait commands: for example, the test case in Lst. 9 would fail if no element with class `.controller` becomes visible within the specified timeout (line 19).

[5] `https://w3c.github.io/webdriver/webdriver-spec.html`

[6] `https://sites.google.com/a/chromium.org/chromedriver/`

[7] `http://mochajs.org/`

Since test runners like Mocha can be run from the command line, it is possible to create command-line scripts that perform automated regression testing on the SUT. These scripts are naturally suitable to be used in combination with a continuous integration system such as GitLab CI[8] or Travis CI[9]. XD-Testing itself has been tested throughout its versions in the GitLab continuous integration environment. A typical scenario of testing in continuous integration works as follows: once the developer commits a new version of the SUT, a command line script is automatically run to deploy the cross-device application (with all of its dependencies) on a testing system. Once the deployment is complete, the script can trigger the test runner, which in turn would start test cases written with XD-Testing.

Finally, while XD-Testing is framework-agnostic, it still supports the integration of specific application frameworks. The idea behind this is that sometimes it is critical to access abstractions and states of the development frameworks used in the SUT. For example, in cross-device applications it may be useful to query the number of connected devices, information usually contained in the internal state of the cross-device framework used for development.

For this reason, XD-Testing allows the definition of facades for individual frameworks that need to be accessed for testing purposes, in a way that their internal state can be accessed from within the test cases. It is even possible to define an adapter to group common functionalities of different frameworks (such as cross-device frameworks) behind a common interface, in such a way that different implementations can be dynamically switched if required. Currently, we have implemented a facade for a cross-device development framework that provides access to various functionalities, such as device pairing, querying connected devices, or handling events defined in the cross-device framework itself.

CASE STUDY

We used XD-Testing to test a cross-device gallery application. The gallery application consists of one controller and any number of viewer devices. The controller device presents a set of thumbnail images. Selecting an image on a controller device shows the image on a paired viewer devices. If more than one viewer is paired to a controller, it should show the subsequent image from the gallery. The controller role is dynamically assigned to the smallest device among the paired devices. In addition, if only a single device is present it will automatically assume both the viewer and controller roles, so that the application is always fully functional.

We have written a number of test cases using Mocha that verify this behaviour and present two of them in Lst. 9. The first one explicitly tests that the application is distributed correctly across devices, specifically it checks that the controller is on the smallest device for a given scenario. This is done by manually adding a phone and a desktop to a scenario (lines 2 to 4). The scenario is then loaded (line 5) and a function of the cross-device framework facade is called to pair the devices (line 6). Then the smaller device of the two is selected with an explicit selector (line 7) and finally the test waits 5 seconds for the controller to show up on the device (line 8). If it does not, the test fails.

The second test case handles the fragmentation implicitly. It checks that an image selected on the controller is always displayed on at least one device. This should hold across any given scenario, making this test suitable for testing with randomly generated scenarios. The scenarios are loaded from the configuration file and, for each scenario, the same test is executed (line 13). The devices are paired (line 16) and the state marked with a checkpoint for the flow recording (line 17). Then, an implicit block is started with the `implicitAny` selector (line 18) which ensures that the following commands are executed on exactly one device. As soon as the device shows the controller (line 19), the second image is clicked (line 20), and the test waits for a viewer to show the image (line 21) which is also recorded with a checkpoint (line 22). The flows generated by executing this test on two and three devices are displayed in the visualiser in Fig. 1.

The first test case benefits from the available device templates that make it easy to quickly instantiate a device with certain properties such as the screen size. It further demonstrates how a specific scenario can be tested with the provided explicit selectors. While the first test case is tailored to one specific scenario, it is relatively simple to add additional devices or exchange a device. The concept of a scenario combined with the device selectors eliminates the need to manually keep track of each device and reduces the required coordination.

The second test case would be even more challenging to write without XD-Testing. Writing this implicit test without the concepts from our DSL would mean that, for every command in the `implicitAny` block, the correct device needs to be identified first and the commands need to be properly synchronised to avoid checking the image before it has been selected. In addition to increasing the amount of code, this would impact readability. Using XD-Testing, the tester can focus on the behaviour while the device selection and synchronisation is handled by the library. Both test cases use the framework integration for pairing devices which facilitate the usage of this commonly used functionality and again makes the tests more concise.

In addition to the two test cases presented here, we have authored tests for checking if additional viewers show subsequent images, if this behaviour also works in corner cases (for example selecting the last image in the list with multiple viewers present), and tests that check the application behaviour when devices connect or disconnect from each other.

Students from our web engineering class had to implement a slightly simplified version of this cross-device gallery as a graded exercise. After they had received the grades, we asked for volunteers to submit their code to us for testing. Our test cases correctly identified issues, mostly related to redistributing the images correctly when devices disconnect.

[8] **https://about.gitlab.com/gitlab-ci/**
[9] **https://travis-ci.org/**

```
1  it('should show controller on the smallest device', () => {
2    let deviceSet = {
3      A: xdTesting.templates.devices.nexus4(),
4      B: xdTesting.templates.devices.chrome() };
5    return devices = xdTesting.multiremote(deviceSet).init()
6      .app().pairDevicesViaURL(baseUrl)
7      .selectBySize('xsmall', small => small
8        .waitForVisible('.controller', 5000)
9      )
10     .end()
11 });
12
13 xdTesting.loadScenarios().forEach(scenario => {
14   it('should show selected image', () => {
15     return devices = xdTesting.multiremote(scenario.devices).init()
16       .app().pairDevicesViaURL(baseUrl)
17       .checkpoint('paired')
18       .implicitAny(device => device
19         .waitForVisible('.controller', 5000)
20         .click('.controller img:nth-of-type(2)')
21         .waitForVisible('.viewer img[src="img/album/large/02.jpg"]', 3000)
22         .checkpoint('image clicked')
23       )
24   })
25 });
```

Listing 9: The first test (line 1 to 11) checks that the controller is displayed on the smallest device. The second one (line 13 to 25) checks if an image selected on a controller is displayed on a viewer device.

DISCUSSION

While our case study demonstrated the benefits of the concepts of XD-Testing, it also demonstrated some of its limitations and directions for future work. First, the explicit selectors are limited to a subset of device properties and only allow selection by one property at a time (such as size). While different types of selectors can be combined via nesting, another option could be to explore a query language for devices which would allow more expressive selectors, for example, *select the smallest device with a keyboard*.

In our case study, we only executed our tests on emulated devices and mainly focused on screen size as the main differentiator. However, there are other factors that could impact a test and thus testing on real devices with different hardware also has its merits. As XD-Testing builds on the standardised WebDriver protocol, it also allows tests to be executed on real devices and could even be used in combination with services like BrowserStack[10] or SauceLabs[11] that offer device farms as a service.

Finally, our visualiser has been optimised for comparing two flows at a time. While it is possible to display more than that, our experience showed that this risks visual overload. However, we would expect a tester to verify more than two scenarios. Visually comparing many different device scenarios at the same time remains a challenge and more options how this could best be presented to the tester need to be explored.

CONCLUSION

We have presented XD-Testing, a library for testing web-based cross-device applications. Our overall aim was to reduce the impact of the fragmentation and address the complexity stemming from the flexibility that is typically present in cross-device applications. To this end, we have introduced concepts for implicitly and explicitly selecting devices and allow tests to be parametrised by device scenarios, which can be randomly generated with our tool. Finally, we provided an option to record and visualise the application flow for informal checks. XD-Testing is lightweight and does not constrain a tester to a particular test runner or cross-device framework. To benefit other researchers and practitioners, we have made it available on NPM[12] and its source code has been released on GitHub[13].

Acknowledgements

This project was supported by grant No. 150189 of the Swiss National Science Foundation (SNF). We would like to thank all our web engineering students who have submitted their code for testing XD-Testing.

[10] **https://www.browserstack.com/**
[11] **https://saucelabs.com/**
[12] **https://www.npmjs.com/package/xd-testing**
[13] **https://github.com/mhusm/xd-testing**

REFERENCES

1. Badam, S. K., and Elmqvist, N. PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proc. ITS* (2014).
2. Bruegge, B., and Dutoit, A. H. *Object-Oriented Software Engineering Using UML, Patterns and Java*. Prentice Hall, 2004.
3. Bruns, A., Kornstadt, A., and Wichmann, D. Web Application Tests with Selenium. *IEEE Software 26*, 5 (2009).
4. Chi, P. P., and Li, Y. Weave: Scripting Cross-Device Wearable Interaction. In *Proc. CHI* (2015).
5. Chi, P. P., Li, Y., and Hartmann, B. Enhancing Cross-Device Interaction Scripting with Interactive Illustrations. In *Proc. CHI* (2016).
6. Dong, T., Churchill, E. F., and Nichols, J. Understanding the Challenges of Designing and Developing Multi-Device Experiences. In *Proc. DIS* (2016).
7. Geronimo, L. D., Husmann, M., and Norrie, M. C. Surveying Personal Device Ecosystems with Cross-Device Applications in Mind. In *Proc. PerDis* (2016).
8. Ghiani, G., Manca, M., and Paternò, F. Authoring Context-dependent Cross-device User Interfaces Based on Trigger/Action Rules. In *Proc. MUM* (2015).
9. Gjerlufsen, T., Klokmose, C. N., Eagan, J., Pillias, C., and Beaudouin-Lafon, M. Shared Substance: Developing Flexible Multi-Surface Applications. In *Proc. CHI* (2011).
10. Hamilton, P., and Wigdor, D. J. Conductor: Enabling and Understanding Cross-Device Interaction. In *Proc. CHI* (2014).
11. Hesenius, M., Griebe, T., Gries, S., and Gruhn, V. Automating UI Tests for Mobile Applications with Formal Gesture Descriptions. In *Proc. MobileHCI* (2014).
12. Houben, S., Tell, P., and Bardram, J. E. ActivitySpace: Managing Device Ecologies in an Activity-Centric Configuration Space. In *Proc. ITS* (2014).
13. Husmann, M., Geronimo, L. D., and Norrie, M. C. XD-Bike: A Cross-Device Repository of Mountain Biking Routes. In *ICWE Workshops* (2016).
14. Husmann, M., Heyder, N., and Norrie, M. C. Is a Framework Enough? Cross-Device Testing and Debugging. In *Proc. EICS* (2016).
15. Kaasila, J., Ferreira, D., Kostakos, V., and Ojala, T. Testdroid: Automated Remote UI Testing on Android. In *Proc. MUM* (2012).
16. Klokmose, C. N., Eagan, J., Baader, S., Mackay, W., and Beaudouin-Lafon, M. Webstrates: Shareable Dynamic Media. In *Proc. UIST* (2015).
17. Leotta, M., Clerissi, D., Ricca, F., and Tonellla, P. Capture-Replay vs. Programmable Web Testing: an Empirical Assessment During Test Case Evolution. In *Proc. WCRE* (2013).
18. Marquardt, N., Diaz-Marino, R., Boring, S., and Greenberg, S. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proc. UIST* (2011).
19. Marquardt, N., Hinckley, K., and Greenberg, S. Cross-Device Interaction via Micro-Mobility and F-Formations. In *Proc. UIST* (2012).
20. Mikkonen, T., Systä, K., and Pautasso, C. Towards Liquid Web Applications. In *Proc. ICWE* (2015).
21. Nebeling, M., Husmann, M., and Norrie, M. C. Interactive Development of Cross-Device User Interfaces. In *Proc. CHI* (2014).
22. Nebeling, M., Husmann, M., Zimmerli, C., Valente, G., and Norrie, M. C. XDSession: Integrated Development and Testing of Cross-Device Applications. In *Proc. EICS* (2015).
23. Rädle, R., Jetter, H.-C., Marquardt, N., Reiterer, H., and Rogers, Y. HuddleLamp: Spatially-Aware Mobile Displays for Ad-hoc Around-the-Table Collaboration. In *Proc. ITS* (2014).
24. Santosa, S., and Wigdor, D. A Field Study of Multi-Device Workflows in Distributed Workspaces. In *Proc. UbiComp* (2013).
25. Schreiner, M., Rädle, R., Jetter, H., and Reiterer, H. Connichiwa: A Framework for Cross-Device Web Applications. In *Proc. CHI EA* (2015).
26. Seyed, T., Azazi, A., Chan, E., Wang, Y., and Maurer, F. SoD-Toolkit: A Toolkit for Interactively Prototyping and Developing Multi-Sensor, Multi-Device Environments. In *Proc. ITS* (2015).
27. Shakeri Hossein Abad, Z., Anslow, C., and Maurer, F. Multi Surface Interactions with Geospatial Data: A Systematic Review. In *Proc. ITS* (2014).
28. Streitz, N. A., Geissler, J., Holmer, T., Konomi, S., Müller-Tomfelde, C., Reischl, W., Rexroth, P., Seitz, P., and Steinmetz, R. i-land: An interactive landscape for creativity and innovation. In *Proc. CHI* (1999).
29. Wozniak, P., Goyal, N., Kucharski, P., Lischke, L., Mayer, S., and Fjeld, M. RAMPARTS: Supporting Sensemaking with Spatially-Aware Mobile Interactions. In *Proc. CHI* (2016).
30. Yang, J., and Wigdor, D. Panelrama: Enabling Easy Specification of Cross-Device Web Applications. In *Proc. CHI* (2014).
31. Zagermann, J., Pfeil, U., Rädle, R., Jetter, H.-C., Klokmose, C., and Reiterer, H. When Tablets Meet Tabletops: The Effect of Tabletop Size on Around-the-Table Collaboration with Personal Tablets. In *Proc. CHI* (2016).