Detecting Responsive Web Design Bugs with Declarative Specifications

Oussama Beroual, Francis Guérin, and Sylvain Hallé

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

Abstract. Responsive Web Design (RWD) is a concept that is born from the need to provide users with a positive and intuitive experience, no matter what device they use. Complex Cascading Style Sheets (CSS) are used in RWD to smoothly change the appearance of a website based on the window width of the device being used. The paper presents an automated approach for testing these dynamic web applications, where a combination of dynamic crawling and back-end testing is used to automatically detect RWD bugs.

1 Introduction

The only functionality of a web application with which the user interacts is via the web page. Today's users expect a lot from a web page: it has to load fast, provide the desired service, and be enjoyable to view on all devices: from a desktop to tablets and mobile phones. However, due to the somewhat complex relationship between HTML, CSS and JavaScript, the layout of web applications tends to be harder to properly specify in contrast with traditional desktop applications. The same document can be shown in a variety of sizes, resolutions, browsers and even devices.

Responsive Web Design (RWD) [15] attempts to provide a solution to this wide diversity, by providing a design methodology that easily adapts the layout to various screen sizes. In RWD, significant portions of a site's graphical user interface can be modified, or even added or removed depending on the specific type of device being used to view a page. However, this appealing feature comes with the drawback that a single web page can now have multiple possible layouts, making the presence of so-called layout "bugs" all the more prevalent. Such problems can range from relatively mundane quirks like overlapping or incorrectly aligned elements, to more serious issues compromising the functionality of the user interface. Detecting these bugs in a responsive application imposes the testing of the interface on all of its possible layouts, which multiplies the testing effort required, when compared to traditional web sites and desktop applications.

It has quickly become clear that detecting GUI bugs in RWD applications requires a new and more efficient testing approach, and especially the creation of testing tools adapted to this specific use case. This is precisely the goal of this paper, which presents an automated technique that provides test oracles capable

of verifying the consistency of a responsive layout over a wide range of window widths. Contrary to existing methods, which define hard-coded algorithms that can test a handful of predefined RWD bugs, our proposed approach defines such bugs as statements expressed in a declarative, domain-specific language designed especially for web interfaces. This language, implemented by the Cornipickle web testing tool [13], includes temporal operators that allow the correlation of elements of a page at multiple moments in time. The novelty of our approach is to leverage this feature by using an external web crawler (in this case, Crawljax [16]) to change a browser's window size multiple times, and instruct the UI oracle to take each of these window sizes as a distinct page. In such a way, we show that RWD bugs can be expressed as specifications over *sequences* of the same page at different sizes.

The important side effect of expressing RWD bugs as declarative specifications is that other types of RWD bugs, currently unforeseen, can easily be detected by simply writing the appropriate declarative specification that corresponds to their occurrence. To the best of our knowledge, this work is the first solution that tackles the issue of responsive web design testing from a purely declarative standpoint.

The rest of the paper is structured as follows. In Section 2, we describe the concept of responsive web design, and describe common examples of RWD bugs. Section 3 describes the current solutions and tools. In Section 4, the paper describes the proposed solution, which consists of combining a declarative language interpreter with a stateful crawler to efficiently detect behavioral bugs in Rich Internet Applications (RIAs). Section 5 shows that with a Cornipickle interpreter as the test oracle for an RIA crawler, one can automatically search and detect behavioral and RWD bugs in web applications.

2 Responsive Web Design Bugs

Before a few years ago, access to websites was conditioned by assumptions about the size of the device's screen. Desktop computers were the dominant device to access websites, and so designers created page layouts that assumed a minimum window size in order to be displayed properly. The situation has changed radically in the past decade, with the advent of smartphones and other devices with smaller screens. A 2019 report highlights that the percentage of internet users in the world via mobile devices and tablets is higher than the percentage of internet users that use desktop computers [11]. An alternative approach for proper site operation in a range of different viewport appliances and sizes was needed.

2.1 Adapting the Layout

A first solution to this problem was to use parameters extracted from HTTP headers: the request for a resource through a browser was followed by a so-called "user agent" string to identify the type of browser used. Reading the user agent string on the server side causes the release of two versions: a mobile version

designed for small screen sizes, and a desktop version designed for large screens. This approach is not without shortcomings. Among its defects, the fact that it does not fit with new devices entering the market, such as tablets that are somewhere in the middle of mobiles and laptops in size, brings the need for another special version of the site. In addition, other versions of the site must be developed in order to satisfy all user devices. Even the assumption that desktop users have large monitors may not always hold: a browser window can share the screen with other applications, and hence not occupy the entire space available. Clearly, a better approach was needed.

Developers have been following the emergence of CSS *media queries* [18], that allow conditional style statements by media properties such as window size. Adapting a site for a specific window size at runtime has become possible, by writing different CSS rules depending on the dimensions of the viewport. Any valid CSS property can be enclosed within a media query, making it possible to enforce very distinct layouts depending on the result of these queries. If CSS rules alone are not sufficient, the standard also defines JavaScript events that are triggered when a media query fires, which makes it possible to write client-side code that reacts to specific changes in a window's dimensions, and to dynamically alter the layout by directly modifying element properties.

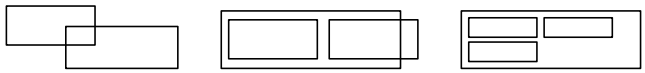
2.2 RWD Bugs

Due to the somewhat complex relationship between HTML, CSS and JavaScript and the multiplication of the devices in the market nowadays, the layout of web applications tends to be harder to properly specify in contrast with traditional desktop applications. The same document can be shown in a variety of sizes, resolutions, browsers and even devices, making the presence of so-called layout "bugs" all the more prevalent. Such problems can range from relatively mundane quirks from elements overlapping to viewport protrusions.

Walsh *et al.* describe five types of bugs in RWD websites [19]:

1. Element Collision: A bug in which elements overlap into one another. This bug can hamper the usability of websites if functional elements of the page are hidden because of this collision. This is shown in Figure 1a.
2. Element Protrusion: Elements need to resize themselves when they are short on space, but they also need be large enough to contain all of their children. Element protrusion is a bug where an element protrudes outside of its parent due to a lack of space. The element can then be unreachable, hidden by another element, or on top of other elements. This is shown in Figure 1b.
3. Viewport Protrusion: This bug happens when elements are pushed outside of the viewport and become inaccessible or hidden. This is shown in Figure 1b, by taking the parent element as the whole viewport.
4. Wrapping Elements: This bug happens when a container is not wide enough to contain all the items and one or more items are pushed on a supplementary line. This is shown in Figure 1c.

5. Small-Range Layouts: Depending on the implementation, some layouts can be correctly displayed in only a small amount of widths. For instance, a display could be only correct between 320 and 325 pixels of width.



(a) Element collision (b) Element/viewport protrusion (c) Wrapping elements

Fig. 1: Various categories of RWD bugs.

3 Existing Solutions

Related works on testing web applications for such kinds of bugs can be roughly divided in two families. The first concentrates on test oracles, i.e. mechanisms for expressing conditions that must be verified by the running application. The second family concentrates on means of finding errors in applications, by performing an exhaustive search of their state space.

There exist a number of tools and techniques for testing web applications. Most of them do not address the functional validation and are not able to test the asynchronous nature and extensive dynamic nature of modern web applications; they are not suitable to test the specific characteristics with respect to Ajax. These tools focus on HTML validation, and static analyses, load testing, broken link detection and protocol conformance.

We first mention web testing software such as Capybara [1], Selenium WebDriver, or Sahi [2]. These tools provide various languages for describing the tests and writing assertions about the application. All these languages are imperative (i.e. procedural), and aimed at driving an application by performing actions. The testing part reduces to the insertion of assert-like statements throughout the script code. By definition, such assertions relate to the current state of the application; they are therefore ill-suited to express bugs that relate multiple states of the application. In fact, in order to express such bugs as assertions, a user must make use of variables to keep data about the state of the application at various moments, and then write assertions in terms of these variables.

Several tools have been developed to provide the service to display a page in a custom window of variable sizes using a web browser. With smart search and quick review features, Websiteresponsivetest [3] supports all major browsers to provide the exact preview of the website on any specific device. Similarly, Respondr [4] allows checking the responsiveness by simply entering the URL of a website. In addition, the device for which a website or web page is tested can also be chosen from the list given. The page can be previewed at an appropriate width.

Screenfly [5] is a multi-device compatibility testing tool which allows previewing web pages as they appear on different devices. Moreover, it supports different screen sizes and resolutions.

The Responsive Web Design Bookmarklet [6] displays any web page in multiple screen sizes for previewing, simulating the viewport of different devices. It is a quick web design tool that can be viewed from a desktop to test any website's responsiveness. All these tools, however, are not automated, and the discovery of RWD bugs still must be done by manual visual inspection of each version of the page. Responsinator [7] helps site owners to get an idea of how their site will work on the most popular devices. Just by typing the website URL, the site will quickly show on screens of various sizes. ResponsivePX's [8] process involves entering the URL of the site and uses buttons to adjust the width and height of the viewport to find the exact breakpoint width in pixels.

Some work has also been done on the use of image analysis techniques to identify layout problems; in particular, WebSee [14] is a tool implemented in Java that leverages several third party libraries to implement some of the specialized algorithms. It applies techniques from the field of computer vision to analyze the visual representation of web pages to automatically detect and localize presentation failures. Applitools Eyes [9] is a commercial tool following a similar principle; it uses the pure image segmentation of the web pages and a pixel-by-pixel visual comparison. However, these approaches are geared towards the detection of static, overlapping or overflow-type bugs in a document, and currently do not support the checking of temporal patterns across multiple snapshots of the same page. We shall see in the following that comparing multiple versions of the same page is key to correctly identify RWD bugs.

The closest tool to our proposed approach is ReDeCheck [20], a responsive web design testing tool. At its core, ReDeCheck builds a *Responsive Layout Graph* (RLG), which accumulates information about the positioning, visibility and relative alignment of an element in multiple versions of the same page. It is inspired from the alignment graph used in X-PERT, a concept that was proposed and developed by Choudhary et al. [10]. ReDeCheck defines three kinds of constraints, respectively called Visibility, Width and Alignment, and reports a responsive layout bug when these constraints differ for an element in the RLG of two versions of the same page. As such, ReDeCheck can only verify a fixed set of predefined layout problems, and does not provide a general-purpose language for expressing assertions.

4 Proposed Solution

In this section, we propose a novel solution for the automated detection of RWD bugs. Instead of requiring the development of special algorithms and a dedicated setup, our approach leverages the combination of two *existing* web testing tools to perform this detection.

The main principle of our approach is shown in Figure 2. First, a tool for driving a browser window is instructed to open the same web page in a web browser multiple times. Typically, the first such call opens the page at a standard desktop window size (w_1), and subsequent calls progressively decrease this width (w_2, w_3, etc.). As one can see in the figure, each distinct size results in a different page layout, with some jumps producing more drastic changes than others (such as the switch from w_2 to w_3). For each of these pages, a summary of the layout is then produced; we call these summaries *page snapshots*. Such snapshots can be created by the web crawler itself, or by some external mechanism (a "probe") fetching the state of some elements of the page and serializing them into some format. The important point is that the succession of such snapshots be kept, in order to form a *sequence* of snapshots.

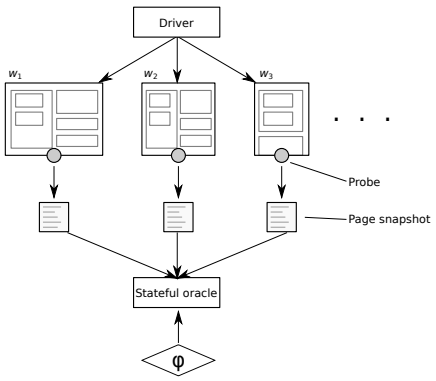


Fig. 2: The proposed framework for catching RWD bugs.

The second part of our approach involves a test oracle, which is fed the sequence of page snapshots, and evaluates a condition, φ, on that sequence. Since our approach involves comparing the state of elements in multiple snapshots across the sequence, the test oracle should be *stateful* —that is, it must be able to handle conditions that take into account the sequence of snapshots. The intuition behind this setup is that a RWD bug will typically be detected as a particular condition on the relative positioning of elements that holds for large window sizes, and which suddenly stops to hold once reaching a smaller width.

This high-level setup constrains the tools that are available for actually implementing the solution. The driver must be able to call a page at different window sizes, and must provide some mechanism for automatically fetching the page's relevant content and produce a summary. On its side, the oracle cannot simply evaluate an invariant condition on each page separately; on the contrary, it must have some form of memory that makes it possible to correlate elements of a page across multiple snapshots. Ideally, expressing these conditions should not be done by writing low-level procedural code (such as pure Java, JavaScript or Python), and allow the user to write RWD bug conditions at a higher level of abstraction for increased modularity and reusability.

Based on these criteria, the solution we propose involves two well-known testing tools: Crawljax [16] as the web driver/crawler component, and Cornipickle [13] as the stateful test oracle. This architecture was coded in an open source plugin for Crawljax[1]. We briefly describe these two components in the following, and explain how they have been made to interact with each other.

[1] http://github.com/liflab/crawljax-cornipickle-plugin

4.1 A Stateful Oracle

The oracle within Cornipickle is on a server that receives requests in JSON format to evaluate a page. These requests are sent by a client browsing the website under test. The developer must inject a JavaScript probe generated by the application into his website to make the requests.

In a standard use case, a developer first writes a set of declarative statements, which are stored in Cornipickle's memory. These statements model the JavaScript code (called probe) that is to be inserted into the application under test so that the client can serialize every page. This probe is designed to report a snapshot of the relevant DOM and CSS data upon every user-triggered event. When such an event occurs, the probe collects whatever information is relevant on the contents of the page into JSON and relays that information to the Cornipickle server, which saves it into a log. Optionally, information on the current status of the assertions being evaluated (true/false) can be relayed back to the probe. An analytics dashboard can then retrieve the saved log and be consulted by the developer, to query the state of all properties input at the beginning of the process.

Cornipickle's language is constructed from first-order and linear temporal logic, such as quantifiers and temporal operators, allowing a user to specify complex relationships on various document elements at multiple moments in time, a feature that is absent from many scripting languages. As a matter of fact, Cornipickle provides operators borrowed from Linear Temporal Logic (LTL) [17] to express assertions about the evolution of a document's content over time. The *Always* x construct allows one to assert that whatever x expresses must be true in every snapshot of the document. Similarly, *Eventually* x says that x will be true in some future document snapshot, and *Next* x asserts it is true in the next snapshot.

One particular purpose of temporal operators is to compare the state of the same element across multiple snapshots. This can be done in Cornipickle with the construct *When* x *is now* y *then* z. If x refers to the state of an element captured in some previous snapshot, then y will contain the state of the same element in the current snapshot.

4.2 Browser Interaction with Crawljax

Crawljax is a tool for automatically exploring the dynamic state of modern web applications. Through programmatic interfaces, it has the capacity to interact with the client side code of the application. The detected changes in the dynamic DOM tree are committed as new states of the behavior. Many options are available with Crawljax to configure the crawling behavior: we can for example specify the links or the widgets to click on or not in the course of the crawling.

This crawler (Crawljax) interacts with Cornipickle through its plugin architecture. Every time a state is created or visited, Crawljax serializes the page and sends it to the interpreter for evaluation the same way the probe sends the page to the Cornipickle server in the traditional architecture. After the page has

been evaluated by Cornipickle, the verdict is returned and our plugin outputs the result.

In order to find RWD bugs, we also created a Crawljax plugin that resizes the browser from a given width to another width. Because having a vertical scrollbar is not a problem in responsive design, only resizing horizontally is the correct approach in discovering RWD bugs. Since we explicitly want to find bugs related to RWD, the plugin slowly lowers the browser's width; these bugs show up on lower widths where the space available becomes increasingly scarce in reference to wider widths. It is possible to provide to the plugin the upper bound, the lower bound and the amount of pixels for the decrement. The plugin also highlights bugs it finds and takes a screenshot of the page. Thanks to Cornipickle's feedback mechanism, the user then gets screenshots where the elements responsible for the bug have red borders.

5 Experiments and Results

In this section, we illustrate how our combination of Cornipickle and Crawljax can be used to automatically detect RWD bugs in websites.

5.1 Defining a Common Language

Cornipickle only provides very low-level access to element properties. Since RWD bugs involve a recurring number of higher-level concepts (containment, overlapping, etc.), it is therefore useful to first define "macro-concepts" that will allow expressing bugs in a more natural way.

The first part of this core constructs defines basic concepts such as alignment and visibility. The definitions are shown in Figure 3. The first statement defines a construct of the form "$x and $y are the same", using the "cornipickleid" property. This property is a unique value given to every element in the page during the serialization phase. Since it is unique, it can be used to identify if two elements are the same across two distinct snapshots of a page. The second statement simply defines a "not the same" construct as the negation of the previous one; it is only added for the sake of readability.

The definition of a visible element checks if its `display` property is set to `none`; invisible elements can be discarded from the analysis, as they do not cause any layout change. Also, this value is affected consciously by the developer so their position on the page is correct. Finally, the alignment of two elements is defined with the constructs "top-aligned" and "left-aligned". We say that two elements are top-aligned and left-aligned when their top and left values, respectively, are equal.

The second part of the core constructs deals with overlapping elements. The corresponding definitions are shown in Figure 4. The first two constructs first define when two elements intersect horizontally and vertically, respectively. Overlapping elements are then defined as two elements that are both visible, and intersect both horizontally and vertically.

```
We say that $x and $y are the
same when (
  $x's cornipickleid equals
  $y's cornipickleid).

We say that $x and $y are not
the same when (
  Not ($x and $y are the same)).

We say that $x is visible when (
  Not ($x's display is "none")).

We say that $x and $y are
top-aligned when (
  $x's top equals $y's top).

We say that $x and $y are
left-aligned when (
  $x's left equals $y's left).
```

Fig. 3: Constructs for visibility, sameness and alignment.

```
We say that $x x-intersects $y
when (($y's right - 1)
    is greater than $x's left) And
  (($x's right - 1)
    is greater than $y's left)).

We say that $x y-intersects $y
when (($y's bottom - 1)
    is greater than $x's top) And
  (($x's bottom - 1)
    is greater than $y's top)).

We say that $x and $y
overlap when (
  (($x is visible) And
      ($y is visible)) And (
    ($x x-intersects $y)
    And
    ($x y-intersects $y))).

We say that $x and $y do not
overlap when (
  Not ($x and $y overlap)).
```

Fig. 4: Constructs for overlapping.

One can see that the first definition uses the expression "`right - 1`", which has for effect that in order to be declared as intersecting, elements should do so by at least two pixels. It overcomes a problem where Cornipickle relays dimensions and coordinates in integers (pixels), although the browser can work with floats in case of elements having dimensions in ratios. These floats are rounded and can cause 1 pixel differences between what is displayed and what is serialized.

Finally, RWD bugs routinely involve the concept of *containment*: the fact that the boundaries of an element are entirely enclosed within the boundaries of another. Containment constructs are shown in Figure 5. These rules define two types of top-level containment: that of a child element within its parent, and also that of an arbitrary element within the browser's global viewport.

5.2 RWD Declarative Properties

Now that we have defined useful concepts at an appropriate level of abstraction, it is possible to express responsive layout bugs as statements using the aforementioned constructs.

```
We say that $c is horizontally
  inside $p when (
  ($c's left is greater than
    ($p's left - 2))
  And
  ($c's right is less than
    ($p's right + 2))).

We say that $c is vertically
  inside $p when (
  ($c's top is greater than
    ($p's top - 2))
  And
  ($c's bottom is less than
    ($p's bottom + 2))).
```

```
We say that $c is fully
  inside $p when (
  If (($c is visible) And
    ($p is visible)) Then (
    ($c is horizontally inside $p)
    And
    ($c is vertically inside $p))).

We say that $x is fully inside
the viewport when (
  If ($x is visible) Then (
    (($x's left + 2) is greater than 0)
    And
    ($x's right is less than
      (the page's width + 2)))).
```

Fig. 5: Constructs for containment.

Scrollbar bug One of the first indications of a poorly responsive website is the presence of a horizontal scrollbar. To detect this bug, a simple Cornipickle property can be defined:

```
We say that there is an horizontal scrollbar
  when (
  the page's width is less than
    the page's scroll-width).

Always (Not (there is an horizontal scrollbar)).
```

This property is made of an auxiliary statement expressing the presence of a scrollbar, which is then used within an LTL temporal operator (`Always`) stipulating that the condition should not appear in any of the page snapshots.

Element Collision The second kind of RWD bug is element collision, which occurs when two elements of the page overlap while they should not. Detecting such bugs is more delicate than it looks. Indeed, it does not suffice to report all overlapping elements inside a page, as many of them overlap for legitimate reasons: to start with, any element nested within its parent would trigger such a simple condition.

This is where the approach we propose, which is based on sequences of snapshots of the same page in various dimensions, can be put to good use. Rather than trying to guess which overlapping elements are suspect by looking at a single rendition of the page, we compare the overlapping state of these elements across successive snapshots. Elements are said to be colliding when they are non-overlapping in one snapshot, and overlapping in the next.

In order to express these properties, one must use the full expressive power of the Cornipickle language, as is shown below.

```
Always (
  For each $x in $(body *) (
    For each $y in $($x > *) (
      For each $z in $($x > *) (
        If ( ($y and $z are not the same) And
          ($y and $z do not overlap) ) Then (Next (
            When $y is now $a (When $z is now $b (
                $a and $b do not overlap)))))))).
```

The three *For each* constructs gather all the elements and their immediate children. It allows testing pairs of *siblings* (elements with the same parent) `$x` and `$y` for their overlap property. The `Next` operator then moves the focus to the next snapshot of the page; the two constructs *When x is now y* trace the same pair of elements and places them into variables `$a` and `$b`, respectively. This way, it becomes possible to compare the properties of a pair of elements over two successive snapshots of the page. Overall, the property says that if two siblings do not overlap at one point in time, these two siblings should not overlap either at the next point in time.

Note that, in the way the property is written, it does not check whether an element overlaps with a "cousin" (an element that shares the same grand-parent): this is not necessary, because a colliding cousin necessarily violates the Element Protrusion property, which we shall describe later. The property could be done by testing every element with every other element but it is costly in performance.

Element Protrusion This property tackles the problem of elements which overflows their container. As with the previous property, reporting all overflowing elements is not appropriate, as overflows can also occur for legitimate reasons. However, one can use the same device, and use LTL temporal operators to compare an element and its direct children across two snapshots of the page. It can be expressed in the Cornipickle language in this fashion:

```
Always (
  For each $x in $(*) (
    For each $y in $($x > *) (
      If ($y is fully inside $x) Then (Next (
          When $x is now $a (
            When $y is now $b (
              $b is fully inside $a)))))).
```

The property at the end has two *For each* constructs that return a pair composed of any element in the page and any of its direct children. Then, if the latter is fully inside the former in an initial screenshot, the same pair should be fully inside in the next one. This property was able to catch a bug on the website `https://www.thelily.com/`. It can be seen in Figure 6 where the *div*

with the menu buttons ends up outside of the menu bar and out of sight. In the first picture, all the buttons are correctly placed in the menu bar. In the second picture, the highlighted "About" button is protruding outside of the menu bar, its parent.

(a) Correct (b) Buggy

Fig. 6: The Element Protrusion bug on the website `thelily.com`.

Viewport Protrusion The Viewport Protrusion bug can be handled in a manner similar to the Element Protrusion bug, but using the whole viewport as the reference. It can be written in Cornipickle as follows:

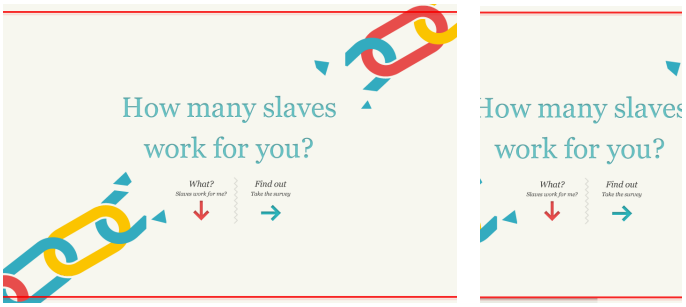
```
Always (For each $x in $(*) (
    If ( $x is fully inside the viewport )
      Then (Next (
        When $x is now $y (
          $y is fully inside the viewport))))).
```

On the website `https://www.slaveryfootprint.org`, a Viewport Protrusion was found in a large width. Figure 7 shows how non-observable bugs can create problems at lower widths. In the first picture, the page's width is already small enough for the document's main *div* element to start protruding outside the viewport. Cornipickle reports it as a bug, although there is not (yet) any observable effect (all the graphical elements and the text inside that div are still completely visible). However, setting the window to an even smaller width makes the bug observable: in the second picture, the window is exactly 440 pixels wide, and we can now see the text overflowing outside the viewport.

Wrapping Elements Wrapped elements are elements that are pushed on an additional line, although they were aligned with other elements on a single line at larger widths. We limited our implementation to elements that are inside a list.

```
We say that the list $x is
aligned when (
  For each $y in $($x > li) (
    For each $z in $($x > li) (
      ($y and $z are top-aligned)
        Or
          ($y and $z are left-aligned)))).

Always (
  For each $x in $(ul) (
    If (the list $x is aligned)
    Then (
```



(a) Correct (b) Buggy

Fig. 7: The Viewport Protrusion bug on the website `slaveryfootprint.org`.

```
Next (                                  When $x is now $y (
                                          the list $y is aligned))))).
```

Finally, all the lists are taken in a first screenshot in order to compare their elements' alignment. They then need to still be aligned in the next screenshot.

An example of a wrapped element can be seen in Figure 8. It could be argued that this is not a bug, however, at lower widths, the list is top-aligned again. This shows that having this list top-aligned is the desired layout.

(a) Correct (b) Buggy

Fig. 8: The Wrapping Element bug on the website `anthedesign.fr`. In the first picture, the list is top-aligned. At a lower width (second picture), the "CGV" element gets pushed on an additional line. The list was highlighted in red by the Cornipickle probe.

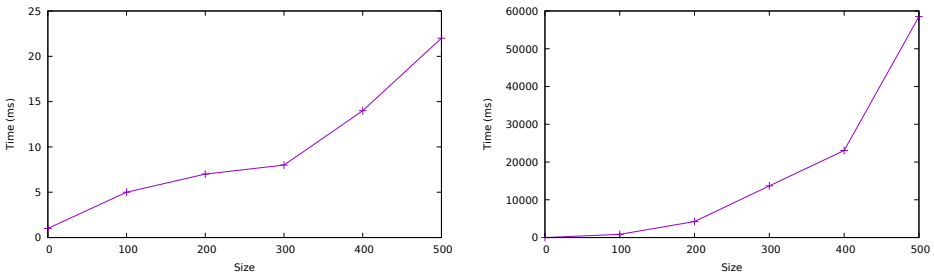
5.3 Scalability Considerations

In order to assess the scalability of our approach on real-world web sites, we created a benchmark designed to measure the computation time of the Cornipickle interpreter on web pages of various sizes. All experiments and data are available as an external download [2], in the form of a self-contained instance of the LabPal experimental environment [12].

[2] `https://github.com/liflab/cornipickle-benchmark`

More precisely, we generated synthetic JSON summaries of pages, in the same format as the one produced by Cornipickle's JavaScript probe. Each page is made of two nested levels of lists, with each list element having a variable number of sub-list elements. Since all properties listed in Section 5.2 compare an element with either its direct children or its immediate siblings, this setup is sufficient to measure the impact of page size on the evaluation of the properties. While using "real-world" web pages seems like an appealing prospect at first sight, static pre-recorded files do not make it possible to run a controlled experiment where parameters can easily be varied. On the contrary, synthetic snapshots allow us to vary the size and structure of pages so that the interpreter's scalability can be measured.

We varied the number of child elements that each list item can have, and ran the Cornipickle interpreter on generated page summaries of the corresponding size. For each snapshot, we measured the total running time of the interpreter for evaluating each of the properties listed in Section 5.2. All experiments have been run on relatively modest hardware, consisting of an AMD Athlon II X4 640 1.8 GHz running Ubuntu 18.04, with a JVM of 3566 MB of memory.



(a) Scrollbar bug (b) Element collision

Fig. 9: Interpreter evaluation time for page summaries of increasing size.

Figure 9 shows the evolution of execution time for increasing page sizes. Due to lack of space, we only include running times for the fastest (Figure 9a) and the slowest (Figure 9b) of all properties. As one can see, checking for the presence of a scrollbar requires a neglgible amount of time on the order of a few tens of milliseconds. The running time is roughly linear in the size of the page snapshot, as it appears that Cornipickle's design requires the ingestion and parsing of every page snapshot, regardless of the amount of data that is actually accessed inside this snapshot.

The running time for evaluating the Element Collision property shows a much larger increase with respect to snapshot size. This is expected, considering the expression of the property as three nested quantifiers (cf. Section 5.2). The first (`$x`) loops over all elements of the page, while the second and third (`$y` and `$z`)

each loop over all children of `$x`. Barring the overhead incurred by the remainder of the expression, a quick calculation shows that the interpreter runs in time $O(m^2)$, where m is the total number of elements in the page; this corresponds to the roughly quadratic execution time we observe experimentally.

To the best of our knowledge, our work is the first to rigorously measure the running time of the Cornipickle interpreter on page snapshots. Unfortunately, the running times we obtained cannot be compared with related works: the paper on ReDeCheck [20], the only other *automated* RWD testing tool, makes no mention of running time on the sample pages it was tested on. The other approaches mentioned in Section 3 all involve a manual inspection, and therefore it makes no sense to speak of running time for these tools. Nevertheless, figures gleaned from [20] can give us a few indications. All pages studied contained fewer than 400 lines of HTML code, and no more than 196 DOM nodes. The experimental results above indicate that pages of such a scale can be handled in under five seconds using our Cornipickle/Crawljax approach.

It shall be noted that our proposed approach is intended to be used in a development and testing context, where tests are run periodically, and a few seconds of waiting is considered reasonable. Performing the same analysis on production web sites in realtime is obviously not an option.

6 Conclusion

In this article we have presented an automated approach that allows the detection of RWD bugs. The effectiveness of the tool has enabled us to catch automatically some common problems encountered in real modern web applications. Cornipickle properties ensure that the pages of an application follow various kinds of constraints. A small application has been developed and integrated in order to test the visual rendering in the different possible viewports in order to catch the RWD faults.

One main advantage of the proposed approach is that it does not require the development of new tools or new algorithms; rather, it leverages the power of two existing systems, and allows RWD bugs to be expressed as declarative test oracle specifications. So far, our solution has concentrated solely on the five types of RWD bugs proposed by Walsh *et al.* [19]; however, the use of a general purpose declarative language opens the door to the elicitation of RWD bugs related not only to layout, but also functionality. We are currently exploring this line of research, which is left as future work.

Our solution also has some limitations. The use of Cornipickle limits us to constraints referring only to elements that are displayed. It makes bugs that are caused by the back end sometimes hard to catch; it is necessary to find displayed elements that can indirectly represent server states. In the same line, if Crawljax does not notify of a state change when the DOM changes, it is not possible to evaluate that page where a bug could have happened. Also, when a property evaluates to false, it is false for the rest of the crawl and no other bug can be caught with this property. This caused a problem with finding observable RWD

bugs because most failures are non-observable and the properties had to find an observable bug as their first bug. Finally, our solution currently does not address cross-browser incompabilities, multi-page analysis, or incoroprate verdicts from other kinds of approaches, such as screenshot-level analysis. Overcoming these limitations could be the basis of future works. A comparison with bugs found by real human testers, as well as ReDeCheck, could be used as a baseline to calculate the precision and recall of our approach.

References

1. http://makandracards.com/makandra/1422-capybara-the-missing-api.
2. http://sahi.co.in.
3. http://www.websiteresponsivetest.com/.
4. http://respondr.io/.
5. http://quirktools.com/screenfly/.
6. https://www.sitepoint.com/responsive-web-design-tool/.
7. https://www.responsinator.com/.
8. http://responsivepx.com/.
9. `http://www.applitools.com`.
10. S. R. Choudhary, M. R. Prasad, and A. Orso. X-pert: Accurate identification of cross-browser issues in web applications. In *Proc. ICSE 2013*, pages 702–711, may 2013.
11. E. Enge. Mobile vs desktop traffic in 2019, 2019. `https://www.stonetemple.com/mobile-vs-desktop-usage-study/`, Accessed July 3rd, 2019.
12. S. Hallé, R. Khoury, and M. Awesso. Streamlining the inclusion of computer experiments in a research paper. *IEEE Computer*, 51(11):78–89, 2018.
13. S. Hallé, N. Bergeron, F. Guérin, and O. Beroual. Declarative layout constraints for testing web applications. *Logical and Algebraic Methods in Programming*, 85 (5):737–758, 2016.
14. S. Mahajan and W. G. J. Halfond. WebSee: A tool for debugging html presentation failures. In *Proc. ICST 2015*, pages 1–8. IEEE, April 2015.
15. E. Marcotte. *Responsive web design*. Eyrolles, 4 edition, 2013.
16. A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (1)*, 6, 2012.
17. A. Pnueli. The temporal logic of programs. In *Proc. FOCS 1977*, pages 46–57. IEEE Computer Society, 1977.
18. F. Rivoal. Media queries – W3C recommendation, 2012. `https://www.w3.org/TR/css3-mediaqueries`.
19. T. A. Walsh, G. M. Kapfhammer, and P. McMinn. Automated layout failure detection for responsive web pages without an explicit oracle. In *Proc. ISSTA 2017*. ACM, 2017.
20. T. A. Walsh, P. McMinn, and G. M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages. In *Proc. ASE 2015*, page 709–714. ACM, 2015.