Web Canvas Testing through Visual Inference

Mohammad Bajammal
University of British Columbia
Vancouver, BC, Canada
bajammal@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—Canvas elements are one of the major web technologies for creating high-performance graphics and visualizations in the browser. The canvas provides APIs for directly painting on the screen, but does not have a DOM state. As such, common web testing techniques that rely on the DOM cannot be applied to canvas elements. Furthermore, there has been little to no research in the literature for testing canvas elements. We propose an automated approach for testing canvas elements and their properties. Our approach performs a visual analysis of the screenshots of canvas elements and infers visual objects, their attributes, and their hierarchical relationships present on the canvas. Each inferred object is then represented as an augmented element inside the canvas element on the DOM tree. Finally, tests are generated from the augmented canvas DOM with assertions that check the inferred objects. We implement this approach in a tool, CANVASURE, and evaluate its accuracy and effectiveness for testing canvas-based applications. Our evaluation results show that CANVASURE has an accuracy of 91% for visually inferring the contents of the canvas, and is capable of correctly detecting 93% of injected visual faults on canvas applications.**

***Keywords*-web canvas elements; web testing; DOM augmentation; image analysis**

I. INTRODUCTION

Canvas elements are one of the major web technologies used to deliver interactive and dynamic graphics-intensive web applications. Canvas-based web applications are utilized in a wide variety of fields, such as math and science education [1], geographical modelling [2], and genetics research [3].

From a testing perspective, however, there has been little to no progress in the literature in terms of testing canvas elements. Existing web testing methodologies do not apply to canvas elements. One of the common approaches of testing web applications in practice is to use a browser automation tool, such as Selenium[1] or PhantomJS[2], to make assertions on the DOM (Document Object Model) of the webpage. This approach opens the web application and analyzes the dynamic DOM tree, and allows developers to write tests that interact with DOM elements and assert their various attributes. However, these DOM-based approaches can not be used with canvas elements. The `<canvas>` element on the HTML page is only a container for graphics, which is updated programmatically, via its APIs, through JavaScript code. It exposes a low-level graphics API, which allows directly painting on the screen pixel-by-pixel, reducing the browser overhead and improving the final real-time speed of the web application. As such, the canvas element does not have a representative DOM-tree for its internal structure and properties that can be tested directly using existing web testing tools and techniques.

An alternative approach of testing web applications is visual testing, using tools such as Sikuli[3] and eggPlant[4]. Visual testing relies exclusively on the application's appearance, with no access to the application's DOM tree. These methods rely on a visual comparison between a number of initial screenshots (i.e., visual locators) created *a priori* by the tester, and comparing them to screenshots at runtime during test execution. Existing visual approaches have several limitations with respect to testing canvas elements, namely, (1) the set of locators (i.e., screenshot images) need to be initially gathered before starting any testing activity. This can be difficult and time consuming, given that each canvas element may be associated with many different screenshots, depending on its dynamic states; (2) the creation of assertions is not fully supported in these tools. More specifically, a visual test consists of a pure image comparison, which does not support the tester in asserting objects on the canvas such as their shape, location, or color, etc; (3) from a maintenance viewpoint, visual testing tools are known to be highly fragile compared to DOM-based methods [4], [5]. Therefore, a large number of visual tests needs to be rewritten for every application version because of the fragile nature of visual analysis used in these approaches—where any minor or inconsequential change (e.g. change of a few pixels) to the image—causes test failure.

In this paper, we propose a novel approach for testing canvas elements that combines aspects of both visual and DOM testing. The approach begins with a visual analysis of the canvas screenshot, infers objects in terms of their shapes, properties, and relationships, and then generates an augmented DOM for the canvas element representing the visual contents of the canvas. This makes canvas elements testable using widely used DOM-based testing techniques, without requiring *a priori* visual locators. In addition, our approach automatically generates tests to check the inferred objects and their properties on the canvas. We implement this approach in a tool called CANVASURE, and evaluate its accuracy and fault detection performance.

Our work makes the following main contributions:

[1]http://www.seleniumhq.org/
[2]http://www.phantomjs.org/
[3]http://www.sikuli.org/
[4]https://www.testplant.com/

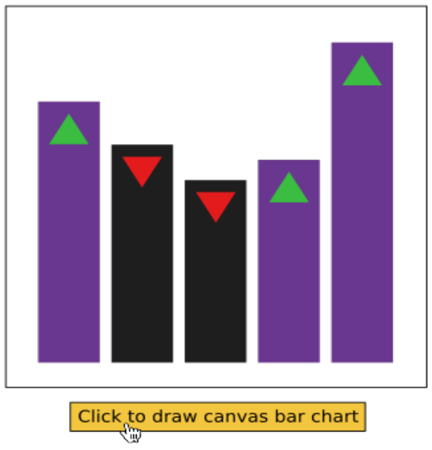

Fig. 1. An example canvas-based application. When the button is clicked, the plot is drawn dynamically (on the client side) using the canvas element.

```
1
2  function onButtonClick() {
3    var canvas = document.getElementById("canvasBarChart
4  ").getContext("2d");
5    ...
6    canvas.beginPath();
7    ...
8    // Example: this would draw one of the
9    // rectangles in the bar chart.
10   canvas.fillRect(leftCoordinate, topCoordinate,
11           width, height);
12   ...
13 }
14
15
16 // The HTML portion of the app
17 <canvas id="canvasBarChart">
18   // This tag remains empty throughout the usage of
19   // the app, due to the lack of DOM representation
20   // for canvas elements.
21 </canvas>
```

Listing 1. JavaScript snippet of the canvas drawing in Figure 1.

1) The first approach for automatically testing web canvas elements through visual analysis, to the best of our knowledge.
2) A technique for inferring objects, their shapes, properties, and relationships from images of canvas elements and representing them as elements in an augmented DOM.
3) An implementation of our approach, called CANVASURE, which supports testing web applications that are entirely canvas-based, as well as web applications that contain only a few canvas elements.
4) An empirical evaluation on 50 canvas screenshots from five canvas web applications. Our results show that CANVASURE has an average accuracy of 91%, and it can detect 93% of injected faults successfully.

II. Motivating Example

Figure 1 shows an example web application that uses canvas elements. A bar chart is drawn in the canvas when the user clicks the button. The corresponding JavaScript code is shown in Listing 1. This example illustrates the dynamic nature of canvas-based web applications. The data is plotted dynamically, on-the-fly, on the client side as opposed to the latency involved in sending the data to be plotted at a server that replies back with an image of the generated bar chart. This dynamic nature makes canvas elements useful in interactive and high-performance visualizations and graphics.

Listing 1 shows a snippet of how the setup and manipulation of canvas elements is done exclusively through the Canvas JavaScript API [6]. The canvas API provides functionality to draw lines, circles, rectangles, etc. These can then be combined and dynamically added to the canvas element in order to draw more complex and interactive drawings. Lines 6 to 11 show a snippet of how the canvas API is used to draw a rectangle on the chart. A call is made to the `fillRect()` function from the canvas API with parameters specifying the rectangle. This allows dynamic creation of shapes on the canvas.

However, the HTML canvas element itself (Listing 1, lines 17-21) remains empty throughout the entire usage of the application. The execution of various canvas API functions does not change or update the contents of the canvas element tag. This is because, as required by the official W3C standard [6] of canvas elements, the canvas is stateless and has no DOM representation. Instead, the canvas API functions are required to draw *directly* to the raw monitor pixels buffer without the costly task of maintaining a DOM state. This direct drawing allows canvas elements to achieve high-speed performance in order to enable dynamic and highly-interactive applications.

Unfortunately, this DOM-free and stateless-behaviour that enables the high-speed of canvas elements is also the reason why they are more difficult to test. At any point during the execution of the application, the state of the canvas remains unknown. While one might conceptually think of gathering state information by tracking the call stack of the API calls, this would not be applicable to an exclusively visual API such as that of the canvas. This is because the actual visual rendered canvas does not directly correspond to the calls made to its API. For instance, calls could mistakenly visually override one another, or they might call the API with unintended arguments, resulting in a wrong visual result. A thousand canvas API calls (for example, draw the rectangle in the example, *one point* at a time) can produce the same resulting visual state as one canvas call (a single call to `fillRect`). Consequently, we can see that it is difficult to assess what state is the canvas in at any given moment which makes it difficult to test canvas elements.

In this paper, we propose an approach that makes canvas elements testable. This approach visually analyzes the canvas screenshot, then creates a DOM tree representing the visual state of the canvas. This makes it possible to test the canvas element using common DOM-testing tools. In addition, the approach also automatically generates tests to check the visual objects of the canvas and their properties.

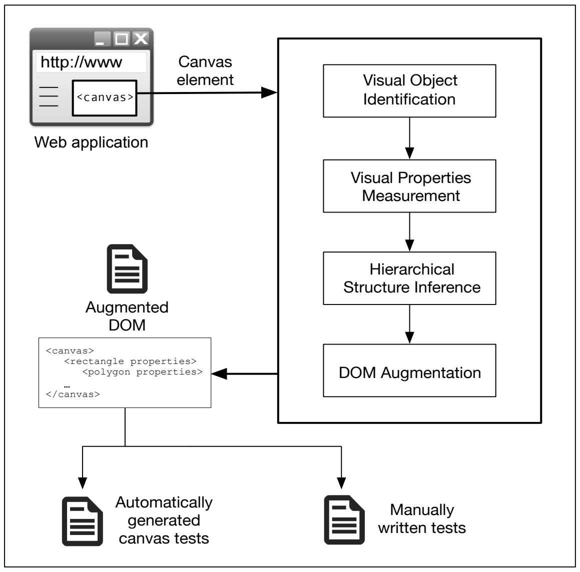

Fig. 2. Overview of the proposed approach.

III. Proposed Approach

The approach starts by automatically opening the webpage containing canvas elements in an instrumented browser. It focuses on canvas elements only and is therefore agnostic of the rest of the web application's page. Therefore, it can analyze canvas elements contained in a larger webpage or canvas applications that are solely composed of a single canvas element.

A. Overview

Figure 2 depicts an overview of the main steps of the proposed approach. For each canvas element on the page, a screenshot is captured and visually analyzed. The visual analysis begins by performing a visual identification of objects on the canvas. Our object identification is capable of identifying common geometrical shapes supported by the canvas API, such as lines, circles, and rectangles, as well as generic arbitrary shapes. Next, the approach infers the visual properties such as color, size, and location, for each of the detected visual objects. A list of all supported properties is shown in Table I. Subsequently, the approach builds a hierarchy information of the visual objects contained in the canvas. This hierarchy information includes parent-child relations between objects, indicating which objects (if any) are contained inside other objects. The hierarchy also includes z-order information, which indicates front-to-back arrangement of overlapping objects.

All the information extracted through the previous steps is then represented as an augmented DOM inside the canvas element. The testing of the canvas is performed by generating assertions from the augmented canvas DOM. We now describe each of these steps in detail in the following subsections.

B. Visual Object Identification

The objective of this stage is to detect and identify objects that are present on a canvas element. We define an object to be present on the canvas if it is rendered and visible in the current screenshot of the canvas. However, the approach does not require an object to be fully visible; occlusions and overlaps of multiple objects are allowed. We apply a number of transformations on a copy of the original canvas image, $\mathbf{C_T}$, in this stage.

Color contrast adjustment. The goal of this first step is to perform a form of color contrast adjustment of the canvas screenshot. This is performed in order to enable our analysis to *see* objects more clearly.

The analysis performs the color contrast adjustment through a *color space conversion* of the canvas screenshot, $\mathbf{C_T}$. A color space conversion changes the representation of the colors of the pixels from one representation system to an alternative representation.

We first need to select a suitable conversion method since several methods exist. To that end, we perform an empirical examination of the eight most common conversion methods [7], namely HSV, L*a*b*, L*u*v*, CIE-RGB, XYZ, YUV, YIG, YPbPr, and YCbCr. Each of these is simply a mathematical formula [7] that changes pixels from one color representation to another.

We empirically evaluate these conversions on a random set of 20 canvas elements.[5] The results of this empirical examination are shown in Figure 3. The figure shows how much contrast there is in the canvas image when using the different color conversions. Higher values are better and indicate more contrast. Based on the results shown in the figure, the YCbCr conversion yields the best color contrast. YCbCr will therefore be our choice for the color space that the canvas will be converted to. Therefore, the algorithm creates a new image, $\bar{\mathbf{C}}_\mathbf{T}$, which represents the canvas screenshot in YCbCr. An example of the outcome of this step is shown in Figure 4-a for a part of the motivating example.

Detecting object boundaries. The goal of this step is to detect the boundary of each object in the canvas. This is performed in order to allow the analysis to roughly estimate where the objects are in the canvas image; this boundary information is used later to identify objects and their properties.

In order to detect these boundaries, we calculate the image gradient [8]. The image gradient is a mathematical processing applied on the canvas image to extract edges, which are the parts of the canvas where there is a transition from one object to another; for example, a boundary between an object and its background.

First, we need to select a suitable method to calculate the image gradient since several methods exist. In order to be able to compute more accurate gradients for various canvas object shapes, we use the Scharr method [9]. We make this choice because this method has been shown [10] to yield good results for smooth and curved objects in addition to angled objects. This would therefore be suitable for processing objects found on canvas elements. We compute the image gradient on $\bar{\mathbf{C}}_\mathbf{T}$, and call it $\nabla\bar{\mathbf{C}}_\mathbf{T}$.

[5]http://corehtml5canvas.com

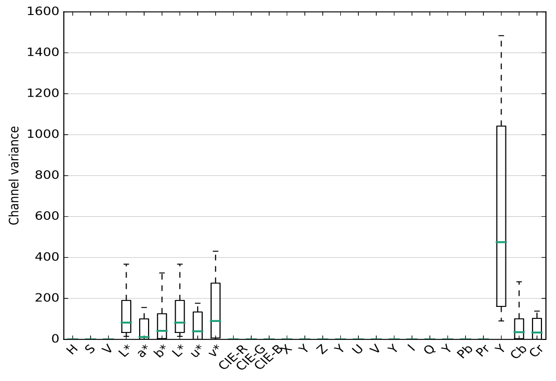

Fig. 3. Parameter estimation for the color contrast adjustment step of the algorithm. Choosing the YCbCr color representation yields best contrast adjustment for the canvas screenshot. Higher values are better.

Next, the analysis performs a binary transformation and converts each pixel in $\nabla\bar{\mathbf{C}}_{\mathbf{T}}$ into either 0 or 1. A value of 1 represents a pixel that is on the object boundary, and a value of 0 means otherwise. The processed canvas image is called $(\nabla\bar{\mathbf{C}}_{\mathbf{T}})^B$. To compute $(\nabla\bar{\mathbf{C}}_{\mathbf{T}})^B$, we need to perform thresholding on the image, i.e., the reduction of a graylevel image to a binary image. We choose a machine learning clustering approach called Otsu thresholding from the computer vision literature, to perform this thresholding. Otsu's method is clustering-based and has been shown [11] to yield high performance. The outcome of this step is depicted in Figure 4-b for the motivating example.

Separating objects. So far, we have a collection of boundaries, but we do not know which group of boundaries belong together as part of the same object. The goal of this step is to separate the different boundaries detected in the last step.

First, our analysis goes through the list of all detected boundaries. For each boundary, the algorithm walks on the boundary, step by step, and examines which other boundaries are connected to it. At the end of this process, we have a group of disjoint boundaries $\mathbf{B}_i$:

$$(\nabla\bar{\mathbf{C}}_{\mathbf{T}})^B = \bigcup \mathbf{B}_i \ : \ \bigcap \mathbf{B}_i \equiv \emptyset \tag{1}$$

where $\mathbf{B}_i$ is a set containing the boundary pixels of the i^{th} detected object in the canvas. The equation shows that each boundary $\mathbf{B}_i$ is separate from other boundaries, $\mathbf{B}_{j\neq i}$.

The algorithm then proceeds by computing the *Euler number* [12] for each boundary $\mathbf{B}_i$. The Euler number is a metric that shows whether a boundary has branches, or is a continuous boundary. For example, geometric shapes such as rectangles or circles have boundaries that continuously surround the shape without branching. On the other hand, two rectangles sharing an edge will have a branch in their boundary.

Next, the algorithm uses the Euler number $\mathcal{E}$ that was just computed in order to check for branching in a boundary. A value of $\mathcal{E} = 0$ indicates that the boundary is one continuous shape without branching. For this case, the algorithm saves the boundary as is without further modifications. On the other hand, a value of $\mathcal{E} < 0$ indicates that a boundary has branching. In this case, the algorithm breaks down the original boundary and creates new boundaries for each branching.

At this stage, the analysis has extracted a set of boundaries $\mathbf{B}_i$ from the canvas image that are separate from each other and non-intersecting. An example of the outcome of this step is shown in Figure 4-c.

Extracting segments on boundaries. The purpose of this step is to help identify the type of each object (e.g., rectangle, circle, arbitrary shapes, etc) based on extracted segments from its boundary. To this end, we take the object boundaries detected so far and break them into smaller line segments. Objects that are curved and smooth, such as circles or generic shapes, are still represented by linear segments too, but as minuscule lines at a zoomed-in scale.

Following the detection of all boundaries $\mathbf{B}_i$, the analysis proceeds by populating each boundary with a random set of probabilistic Hough segments [13]. These segments are small linear portions of the boundary of each region. This approach has been shown [14] to produce robust extraction of boundary segments. In essence, this step performs rough clustering of boundary coordinates and groups neighboring points into a set of small segments.

The generated probabilistic segments are, however, often redundant duplicates, overlapping, and/or intersecting. This presents a challenge in terms of identifying which segment is a true linear segment and which is redundant. As such, the set of all generated segments does not have a direct 1-to-1 correspondence to linear segments on the boundary.

We address this by performing a post-processing of the generated segments. Post-processing starts by creating an *R-tree* index. An R-tree [15] is a database structure that allows storing, querying, and retrieving data that is spatial in nature (e.g., locations, point sets). Our set of segments is also an example of such spatial data. Therefore, we insert the segments into an R-tree index in order to efficiently perform spatial queries such as finding intersecting segments or neighbors.

Accordingly, two sets of operations are performed on the R-tree index. The first operation consists of iterating through each line segment in the index and then querying the existence of other segments with full or partial spatial intersection. The second operation also iterates through each line segment in the index, but queries the existence of neighbouring segments instead of intersections.

For the first R-tree operation, the result of each iteration is a set of segments that are fully or partially intersecting. The algorithm then performs a pair-wise iteration over the intersecting subset. For a pair of line segments that are parallel, the algorithm removes the smaller segment if it is fully enclosed in the larger segment. Otherwise, in cases where the rectangles are partially overlapping, the two parallel segments are merged into one new line segment, and then removed from the set.

For the second R-tree operation, the result of each iteration is a subset of segments that are neighbours of each other. The algorithm forms this subset by querying the R-tree index for the 4 spatial neighbours, one neighbour in each direction, around the query rectangle. The algorithm then performs a pair-wise iteration over the subset. Pairs that are both collinear and parallel are then merged into one new segment, and the forming segments are dropped from the set. Other pairs are left in the set as is.

At this point, we have constructed a final set of line segments for each boundary. The analysis is ready to identify the shape type (e.g., lines, rectangles) of each object. An object whose set of segments has a cardinality of 1 is reported as belonging to the 'line' type. Similarly, a set of three lines are triangles, and four lines are rectangles. For other objects, the analysis first performs an ellipse fitting to cover the object. Then, the relation between the major and minor axes of the ellipse is computed. If these axes are equal, then the detected shape is identified as a circle if the object's set of line segments has a cardinality of more than 4 (i.e. more than a rectangle) and its area is *solid*. A region is defined as solid when the the area of that region is identical to the area of the convex hull of the same region. On the other hand, when the region is not solid, the object is identified as a polygon.

Table I lists the different object types recognizable by our approach. An example of the processing performed in this step is shown in Figure 4-d.

C. Visual Properties Measurement

At this stage, our technique measures the visual properties of the objects detected in the previous section. These properties are extracted from the original canvas screenshot and include parameters such as location, size, and color. Table I shows a list of the visual properties that are measured by the technique for the detected visual objects. This subsection describes the process by which we measure these properties.

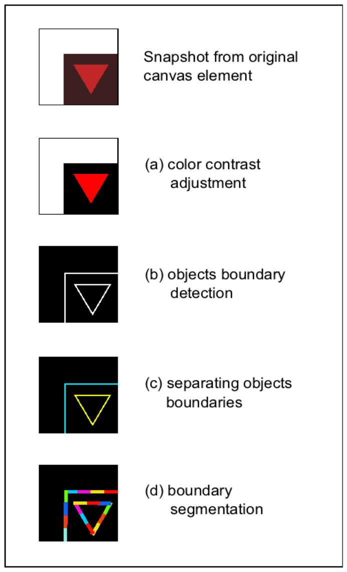
Snapshot from original canvas element
(a) color contrast adjustment
(b) objects boundary detection
(c) separating objects boundaries
(d) boundary segmentation

Fig. 4. Illustration of the general stages involved in the visual identification of canvas objects. Best viewed on a color monitor.

Object position. First, the technique calculates the centroid of each detected object. This is done by computing the arithmetic mean of all coordinates in the object. The centroid would therefore represent the point that minimizes the Euclidean distance between itself and other coordinates in the object.

Color. Next, for all coordinates in the object, the analysis computes the median pixel vector. This is performed as to remove outliers that might be present due to small protrusions in the detected region. This vector is then assigned to the color property of the detected object.

Size and orientation. Subsequently, the analysis performs an ellipse fitting to the region of the object. The ellipse is centred on the centroid, and its axes are fitted so that the ellipse covers the entire object. By the end of this process, an ellipse is generated for each detected object, with each ellipse having a centre, a major axis, and a minor axis. The analysis then extracts a number of visual properties based on the fitted ellipse. First, if the shape is a rectangle, the ellipse axes determine the width and height. If the shape is a circle, then ellipse axes determine the diameter. Next, an orientation is assigned to the object. This orientation is computed as the angle between the major axis of the ellipse and the positive horizontal axis. This represents how much the object is rotated.

Point set. Finally, the analysis extracts a representative point set for each object. This point set lists the coordinates necessary to reconstruct the object. This set is only generated for triangles and generic shapes (polygons), since other object categories are fully identified using their other measured properties, as shown in Table I.

In order to generate this representative point set, the analysis takes as input the coordinates of the object boundary and then proceeds to apply the *Harris* operator [16], which is a mathematical transformation that detects points of intersections in boundaries, as follows:

$$H = \sum_{u} \sum_{v} w(u,v) \begin{bmatrix} C_x^2 & C_x C_y \\ C_x C_y & C_y^2 \end{bmatrix} \tag{2}$$

where C_x, C_y are the partial derivatives of the canvas screenshot, and $w(u,v)$ is a shifting window throughout the image. This process extracts points with more than one derivative direction, which yields points where linear segments intersect.

D. Hierarchical Structure Inference

For each detected canvas object, the analysis then performs a sequence of operations on the canvas screenshot that we have processed so far in order to infer any hierarchical information. We define hierarchical information as the information pertaining to the relative spatial relations between overlapping or nested objects. Objects that are not overlapping or nested have no hierarchical information.

More specifically, the hierarchical information of an object has two parts: the z-order and the parent-child relations. The z-order determines the relative arrangement of overlapping objects along the z-axis (i.e., front to back arrangement). While the width of the screen is represented with an x axis and the

TABLE I
LIST OF THE INFORMATION THAT IS VISUALLY INFERRED FROM THE SCREENSHOTS OF WEB CANVAS ELEMENTS

Visual object identity	Visual properties	Hierarchical structure
lines, triangles, rectangles, squares, circles, generic shapes (polygons) * Note: our approach allows multiple overlapping, partially visible, or nested combinations of these objects.	- centroid: geometric center point of objects - size: - width and height: for rectangles and squares - diameter: for circles - color (HTML hex values) - orientation: number of degrees of rotation from horizon - point-set: the set of points representing generic shapes (polygons)	- z-order: relative order of objects in the z-direction (i.e. behind or in front of other objects) - Parent-child relation: which objects are contained inside other objects.

height of the screen represented by a y axis, the z-axis is perpendicular to the xy screen plane and points towards the user. That is, it determines which objects are closer to the viewer and away from the screen, and which are away from the viewer into the screen.

The second part of hierarchical information is the parent-child relations. This hierarchical information examines nested objects and determines which are children objects and which objects are their parents. We define object A to be a child of object B if object A is entirely visually contained inside object B. Equivalently, object B is a a parent of object A if it entirely contains it. Objects that are not fully contained in other objects (for example, in cases of partial overlaps) are siblings, and do not have a parent-child relation.

The analysis starts the inference process by creating two directed acyclic graphs for the detected objects. The first graph, $\mathbf{P}$, captures parent-child information. The second graph, $\mathbf{Z}$, contains z-order information. Every object is initialized as a degree zero vertex on the graphs. Subsequently, the analysis creates an R-tree index. Each item in the index is a 2-tuple of a detected object and its minimum bounding rectangle.

The analysis then iterates over the collection of detected objects. For each object, the analysis performs a spatial query on the R-tree to determine the leaves that are intersecting with the object. This results in one of three possible scenarios. The first is when the R-tree returns no intersections with the object. The second is when the intersecting object has its minimal rectangle fully inside the query object. The third case is when the intersecting rectangles are partially overlapping.

For the first case, the minimal rectangles of the tree leaves are non-intersecting. This query result indicates that there are no overlapping or nested objects with the current object. As such, the object has no defined hierarchical information. The $\mathbf{P}$ and $\mathbf{Z}$ graphs are therefore not updated, and the corresponding vertices of the object remain in their initial state of zero degree.

For the second case, the query indicates that one or more objects are fully contained inside the query object. Therefore, this constitutes a parent-child relation. Accordingly, we update both graphs $\mathbf{P}$ and $\mathbf{Z}$. The update to $\mathbf{P}$ is a new directed edge from the child to the parent. Similarly, the update to $\mathbf{P}$ is also a directed edge from the child to the parent.

Finally, for the last case, the query returns a partial overlap of the minimal rectangles. As such, this case would not constitute a parent-child relation. However, it is still possible for this case to have z-order relations. Accordingly, the analysis proceeds to detect the presence of a z-order relation. First, the analysis generates the concave parts of the region. These parts are the subtraction of the convex hull of the region from the original region. Next, the analysis performs an element-wise logical XOR between concave parts of the object and the objects with the partially overlapping minimal rectangles. If the result of the operation is a non-zero region, then a z-order relation exists. The analysis proceeds by creating a directed edge in $\mathbf{Z}$ from the object with the concave parts to the object that is resulting from the R-tree query.

E. Testing through DOM Augmentation

At this stage, the analysis has concluded the process of building information about the identity, properties, and hierarchical structure of the objects on the canvas. The analysis proceeds by casting this information into a DOM tree and augmenting it to the original canvas element.

DOM Augmentation. First, the analysis creates a DOM node for each detected object on the canvas. The tag name of the node matches the identity of the object. For example, an object that has been identified as a triangle is represented as `<triangle>`, generic shapes (polygons) as `<polygon>` and so on.

Next, for each node in the DOM, the analysis inserts all the detected attributes pertaining to the identified object (see Table I for a full list of attributes). For example, a `diameter` attribute is added to each `<circle>` node.

Finally, the analysis then proceeds by arranging the nodes of the DOM according to the inferred hierarchical information. We recall that the hierarchical information consisted of two parts: the parent-child relations contained in the $\mathbf{P}$ graph, and the z-order relations contained in the $\mathbf{Z}$. For the parent-child relation, the analysis re-orders the nodes of the DOM as to make child nodes in the DOM correspond to child objects on the canvas (as described in the hierarchical structure extraction section). The analysis then adds a `z-order` attribute to each node. A z-order of zero indicates a front-most object, and elements further back have increasing negative values.

At this stage, the analysis has finished constructing an augmented DOM tree for the canvas element. An example is shown in Listing 2 representing the generated augmented DOM corresponding to the motivating example. As shown in lines 4 and 8 of the listing, a triangle node is located inside a rectangle node in the DOM. This corresponds to the state of the original canvas in Figure 1, where one can see that a triangle is visually located inside a rectangle.

Testing process. The proposed canvas DOM augmentation approach was designed to extend and enable the use of DOM testing techniques with canvas elements. As such, it can be used in any testing process that is based on the DOM. For example, through browser automation using Selenium, a test can be written to navigate to a page, click a few buttons, and then finally, through the proposed augmentation approach, assert that the canvas has a vertical red rectangle, for instance. This is similar to the common task performed in DOM tests when the presence of a certain element, say a `<div>` with a certain id, is asserted.

The inferred augmented DOM is then used to test the canvas element in one of two ways. In the first testing approach, a list of assertions can be automatically generated by default to assert the presence of all nodes of the DOM and their attributes and hierarchy, as shown in Listing 3. The generated assertions are broken down into three types of assertions that correspond to the information inferred from the canvas: (a) assertions on the identity of objects, (b) assertions on the properties of objects, and (c) assertions on the hierarchy of objects. The reason for separating assertions into three different layers is twofold. First, this enables pinpointing the cause of assertion failures, as opposed to writing one assertion that asserts the state of the entire canvas as a whole. Second, this strategy gives the user a flexible way of choosing which aspect of the canvas to test. For instance, the user can indicate that they are only interested in testing the presence of objects on the canvas, regardless of position or color. Of course, the user can keep the default option where all attributes are tested. Therefore, instead of just simply reporting that a test has failed, the approach, for example, can report that the test has failed because a triangle was absent or has changed color in the canvas.

Alternatively, in the second testing approach, the augmented DOM can be used in a case where a tester would write specific tests to assert the existence of particular objects, properties, or certain hierarchies of interest. This use case would, therefore, be more appropriate for scenarios such as test-driven development, for instance.

F. Tool Implementation: CANVASURE

We implemented our canvas visual inference approach in a tool called CANVASURE [17]. CANVASURE is implemented in Python 3. We use the numpy [18] library to import basic mathematical and numerical functions, and use the scipy [19] library for matrix computations on the screenshots. We use the Selenium web driver to run web applications and extract canvas elements.

IV. EVALUATION

In order to assess the accuracy and effectiveness of CANVASURE, we examine the following research questions:

RQ1: How accurate is CANVASURE in visually inferring canvas elements and their properties?

RQ2: How effective is CANVASURE in detecting faults in canvas elements?

A. Subject Applications

Our main criterion for selecting subject applications was the central role of canvas elements in the function of the application. More specifically, the applications should either be entirely canvas-based, or use canvas elements as the main and central display of information. This selection criterion helps in obtaining canvas elements that are more rich and complex, since they represent the central component of the application. Note that CANVASURE is completely agnostic to the rest of the web application and can therefore process a single canvas element on its own or canvases that are central displays of the entire application. Using this selection criterion, we were able to collect as much as five open-source applications that use canvas elements. A list of these applications is shown in Table II. As can be seen in the list, the applications cover a variety of sizes, ranging from between 7,200–105,000 lines of code. The applications cover a variety of domains, including graphics, medicine, chemistry, and music.

B. Experimental Procedure

Accuracy: RQ1. Through RQ1, we aim to evaluate the accuracy of CANVASURE in visually inferring objects from the canvas. This is important in order to ensure the inferred augmented canvas DOM exhibits a faithful representation the visual canvas. To this end, for each subject application, we collect a random sample of 10 canvas screenshots, for a total of 50 screenshots of canvases from the 5 subject applications. Before the screenshots of the canvases are taken, the code temporarily makes all non-canvas elements invisible, such as advertisement boxes or other superimposed areas, in order to make sure that the snapshot is specific to the canvas element. We also note that the number of canvas elements is not that same for all subject applications, or at different points throughout the use of the same application. As such, when conducting the evaluation, by temporarily hiding all non-canvas elements the screenshot is taken from the page without specifying the canvas element. In a production environment, the developer would of course have the option, if needed, to specify which element to test.

We then generate the augmented canvas DOM for each collected canvas screenshot. Subsequently, we recreate a canvas image from the augmented canvas DOM. Finally, we compare the similarity between the snapshot of the original

TABLE II
LIST OF WEB APPLICATIONS USED FOR EVALUATION

Application	Description	LOC
JBrowse [20]	Genome browsing and visualization	105,427
Reactome [21]	Reactions analysis	27,702
Scribl [22]	Genetics analysis	20,939
Gibberish [23]	Music composition and production	14,884
iCanplot [24]	Data plotting and visualization	7,269

```
<canvas id="canvasBarChart">

 <rectangle center="(191,43)" width="86" height="295" z-order="-1" color="#673C8C">
   <triangle center="(107,43)" point-a="(93,43)" point-b="(114,33)" point-c="(114,55)" z-order="0" color="#41A74D"/>
 </rectangle>

 <rectangle center="(166,264)" width="87" height="355" z-order="-1" color="#673B8C">
   <triangle center="(57,264)" point-a="(43,265)" point-b="(64,254)" point-c="(64,276)" z-order="0" color="#41A74D"/>
 </rectangle>

 ... the rest of the generated DOM ...
</canvas>
```

Listing 2. The visually inferred augmented DOM for the canvas element of the motivating example (Figure 1 and Listing 1)

```
// High-level test: testing the identity of objects in the canvas
 assertTrue("'canvasBarChart' should contain 5 triangles",
   webDriver.findElements(By.xpath("//canvas[@id='canvasBarChart']//triangle")).size() == 5);

 assertTrue("'canvasBarChart' should contain 5 rectangles",
   webDriver.findElements(By.xpath("//canvas[@id='canvasBarChart']//rectangle")).size() == 5);
   // ...

// More detailed test: testing the locations of objects in the canvas
 assertNotNull("'canvasBarChart' should contain a triangle at (107,43)",
   webDriver.findElement(By.xpath("//canvas[@id='canvasBarChart']//triangle[@center='(107,43)']")));
  // ...

// More detailed test: testing the size of objects in the canvas
 assertNotNull("'canvasBarChart' should contain a rectangle with width=86 and height=295",
   webDriver.findElement(By.xpath("//canvas[@id='canvasBarChart']//rectangle[@width='86' and @height='295']")));
  // ...

// ... Tests for other properties (z-order, color, etc)
```

Listing 3. The automatically generated test assertions for the canvas element of the motivating example (Figure 1 and Listing 1)

visual canvas element, and the canvas image created from the augmented canvas DOM.

We compute the accuracy as a normalized root-mean-square (RMS) error ΔE in order to obtain a normalized similarity score to enable comparison across subjects. This accuracy measures the pixel-by-pixel similarity between C_O, the original canvas screenshot image, and C_{DOM}, the canvas image reconstructed from the augmented DOM. We compute this accuracy measure as follows:

$$\Delta E = 1 - \frac{\sqrt{\frac{1}{n}\sum(C_{DOM} - C_O)^2}}{\| (C_{DOM} - C_O) \|_2} \tag{3}$$

a ΔE value of 1.0 (i.e., 100%) indicate high accuracy (an identical match), while lower values indicate lower accuracy.

Effectiveness: RQ2. The objective for addressing RQ2 is to assess the effectiveness of the tool in terms of its fault detection ability. To this end, for each collected canvas screenshot, we run CanvaSure to infer the augmented canvas DOM. Subsequently, we inject random modifications into these canvas screenshots and generate another augmented canvas DOM. This process would therefore yield two DOMs: an original pre-injection DOM, and a post-injection DOM.

The fault injections take the form of injecting a random shape with a random set of points and random attributes (e.g., color, size, etc) directly on the canvas screenshot. The rationale for this fault injection model is to have the same degree of randomness across all evaluated applications, which can be difficult to ensure due to the following factors. First, the API of each subject application has varying degrees of being able to mutate the final visual state of the canvas. For example, some allow direct modification of the final visual content on the canvas, while for other applications the API allows only one or two properties to be changed. Furthermore, the percentage of code contributing to the canvas visual state varies across subject applications. In other words, some applications have a large core of business logic code relative to only a small part of codebase for canvas drawing, while for other applications the code is almost purely visual code for canvas drawing. Accordingly, these differences add a confounding factor to the evaluation and skew the performance of different applications relative to others. We therefore adopt the fault injection approach outlined above in order to have a more unbiased evaluation.

We therefore proceed as follows. We begin with the 50 canvas screenshots collected from the 5 subject applications. For each screenshot, we perform two injection runs, with one automatic random injection per run. Therefore, we end up with a total of 100 random injections for the 50 canvas screenshots collected from the 5 subjects.

The fault detection performance is then measured using the Jaro-Winkler [25] similarity ΔS between the pre-injection DOM and post-injection DOM. We chose this metric because it provides a normalized score, which would facilitate comparison and reasoning about results. For the purposes of this evaluation, we use the DOM instead of test assertions for

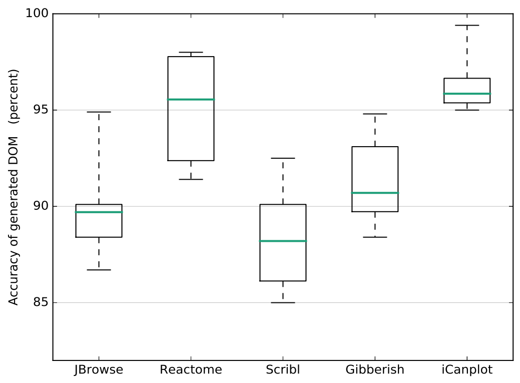

Fig. 5. Accuracy of the inferred augmented DOM for each evaluated application in table II. A total of 50 canvas elements in five different applications where used for this experiment.

two reasons. First, the DOM is the source from which the assertions themselves are generated (as explained in section III-E). We therefore evaluate the fault detection performance more accurately by checking the source DOM itself. Second, using a canvas assertion for this evaluation would introduce a bias as it requires a subjective human evaluation of whether a true positive or false positive actually occurred (e.g. does this object actually look bigger than before?). For these reasons, we perform a direct DOM distance comparison to avoid these biases and conduct a more accurate quantitative evaluation.

Accordingly, we define a true positive result and a false negative result using the Jaro-Winkler similarity ΔS between the pre- and post-injection DOMs. A true positive is defined as the case when the tool detects the injection. A false negative is defined as the case when it does not detect the injection. A false negative corresponds to $\Delta S \equiv 1$, where the pre- and post-injection DOMs are identical. Alternatively, a true positive corresponds to $\Delta S < 1$, where a difference was detected between the DOMs.

C. Results and Discussion

Accuracy. Figure 5 shows box plots for the accuracy of CanvaSure. The x-axis shows our subject applications, and the y-axis shows the similarity percentage between the original canvas and the canvas regenerated from the inferred augmented canvas DOM. The runtime average for the tool is relatively fast at around 4 ± 0.3 seconds.

We make a number of observations from figure 5. First, we note that most inferred DOMs have an accuracy close to or above 90% (the average for the dataset is $91.2\% \pm 1.3$). Furthermore, at the highest and lowest of outliers, the accuracy ranges between 98% and 85%. Therefore, we conclude form these results that the DOM inference process is relatively accurate in representing the canvas.

As for the accuracy at the lower end of the outliers, the reason for this is the inability of the probabilistic Hough transform used by the algorithm (as described in section III) to generate segments to cover the small boundaries of small objects. These often take the form of small appendages connected to various objects, and the probabilistic Hough transform misses such small appendages. In a future version of the algorithm, we plan to improve the performance by finding a more suitable alternative than the probabilistic transform.

Effectiveness. Table III shows the effectiveness results for CanvaSure. Each row shows the total number of faults injection performed on the application, and then the number of true positives and false negatives reported by CanvaSure, as described in section IV-B.

From Table III, we note that the overall rate of detecting true positives is 93% and the overall rate of false negatives is 7%. Furthermore, we note that the per-application true positive rate varies from 16 out of 20 (80%, lowest) to 20 out of 20 (100%, highest). As such, we conclude from these results that the approach is relatively reliable in detecting faults in canvas elements.

The main reason for the false negatives is that the probabilistic Hough transform used by the algorithm was unable to generate segments that would cover some of the injected shapes that were small appendages to other objects and this resulted in no modification to the segments set of the object. This is therefore reflected as a false negative. As part of our future improvements, we would like to improve the true positives rate even further by finding a better alternative to the probabilistic transform.

Threats to Validity. Our choice of subject applications could be a source of external threat to validity. In order to mitigate this threat, we select subject applications from a wide range of complexity (from around 7,000 to more than 100,000 LOC), and also select the applications from a varying assortment of fields, such as medicine, music, and chemistry. Another related threat is the relatively low number (five) of evaluated subjects. We mitigate this threat by extracting a random collection of 50 canvas snapshots for these applications and running 100 fault injections. Furthermore, variations in the fault injection process could be a source of internal threat to validity. We mitigate this threat adopting a uniform approach of consistently injecting random objects on the canvas regardless of the applications's API, as opposed to individually injecting faults by modifying the specific API parameters of each canvas app. This helps in ensuring equally random fault injection across subjects. Another related threat is the possible bias in the determination of true positives or false negatives. We mitigate this threat by injecting faults across all trials and then using a

TABLE III
Results of detecting fault injections

Application	# Injections	# True Positives	# False Negatives
JBrowse [20]	20	18	2
Reactome [21]	20	20	0
Scribl [22]	20	16	4
Gibberish [23]	20	19	1
iCanplot [24]	20	20	0
Total:	100	93%	7%

quantitative distance metric with a normalized range to allow a direct binarization of results into true positives and false negatives.

Another potential threat to validity is the issue of cross-browser compatibility. We note that the rendering of canvas elements is categorically different from rendering regular web pages. When rendering a page, the browser has to continuously maintain a DOM and regularly recompute layout position and visual properties of the page elements before rendering. For canvas, however, the browser does not maintain any layout information about the canvas, and simply paints the pixels from the API call directly on screen. All browsers relay the pixel-by-pixel rendering directly to the canvas. Canvas elements are stateless objects and browsers do not participate in maintaining what does or does not get rendered. This is the opposite of rendering webpages.

To the best of our knowledge, there is no evidence in the literature of cross-browser incompatibilities for canvas elements. Furthermore, in a recent study [26] categorizing questions asked by developers on StackOverflow, canvas related questions were mostly about how to use the canvas API. There was no reported pattern of questions on canvas-related cross browser incompatibility issues. In addition, and more importantly, from our own use and testing of canvas elements, we did not notice incompatibility issues either. For these reasons, we do not consider cross-browser incompatibility to be a threat to the validity of the proposed approach.

V. Related Work

There has been little to no research in the literature regarding testing canvas applications. Due to the absence of an object model for canvas elements, it is difficult to apply conventional web testing techniques to canvas elements.

Nonetheless, there exist a few open-source attempts, which can be used to perform basic testing of canvas elements. However, these open-source tools are not part of a research publication, and therefore they lack detailed experimentation or a thorough explanation of the methodology. We briefly discuss some of these tools.

Canteen[6] takes a callstack analysis approach. The tool captures a stack of the function calls sent to the canvas element, then compares the runtime stack against a known correct call stack manually provided by the developer. A major disadvantage of this approach is that it focuses exclusively on the call stack, meaning a test's pass or failure does not directly correspond to the visual state of the canvas. That is, the actual rendered canvas that the end user observes is not tested.

Other open-source tools, such as Needle[7] and JS-ImageDiff[8], adopt a visual approach instead of code analysis. Runtime screenshots are compared against known good screenshots provided by the test writer. However, being a direct image differencing approach, it shares much of the fragility issues [4], [5] of visual approaches, where even a single pixel could result in test failure. Furthermore, this approach still requires test writers to provide *a priori* visual oracles.

There is another related body of literature that is related to visual-based approaches for testing web pages in general, but not for canvas testing. For example, WebDiff [27] detects cross-browser incompatibilities for a given webpage. It performs an indirect comparison of the appearance of the webpage in two browsers using DOM-based analysis. Although the tool subsequently confirms the initial DOM-analysis using a visual comparison, the approach still requires the DOM to detect the cross browser incompatibility in the first place, and uses a visual comparison as a confirmation. Although the tool was shown to be helpful in terms of general web page cross-browser testing, it cannot be used with canvas elements because they lack a DOM representation.

Another approach, using the WebSee [28] tool, targets the problem of detecting and locating visual inconsistencies in web applications. The approach uses visual comparison to detect visual differences between pages, and then locates the inconsistency using DOM elements. While the approach showed good performance in detection and localization of inconsistencies, it does require the DOM in its operation and therefore can not be used with canvas elements.

PESTO [29] aims at simplifying the creation of visual web tests. The approach starts with a given DOM-based test suite and then generates visual locators. It then concludes by generating a visual test suite that matches original DOM test suite. While it shows good performance, canvas elements can not be used with this tool because they do not have a DOM.

Scry [30] focuses at assisting developers in explaining and reproducing visual changes on a web page. It asks the developer to specify which webpage element to monitor, and then watches that element. Whenever the appearance of the element changes, the method stores the DOM and CSS of that element in order to determine the code changes that produced the appearance change. While this tool has interesting applications and could help explain appearance changes in a web page, it can not be used for canvas elements because it is based on analyzing the DOM of elements. Canvas elements, however, have no DOM-tree representation, which makes it incompatible with such DOM-based tools.

VI. Conclusions

Web applications based on canvas elements allow the creation of dynamic graphics, interactive user interfaces, and scalable visualizations. However, there has been little to no research in literature in terms of testing canvas elements. This paper proposed a testing approach, implemented in a tool, CanvaSure, based on visual analysis of the screenshot of canvas elements, and generating an augmented DOM tree for the canvas element to allow making test assertions on it. We evaluated the accuracy of the proposed approach and its effectiveness in detecting faults injected in canvas elements. We found the inference process to be relatively accurate (around 91% accuracy on average) with a true positive rate of 93% in detecting fault injections.

[6]https://github.com/platfora/Canteen
[7]https://github.com/bfirsh/needle
[8]https://github.com/HumbleSoftware/js-imagediff

References

[1] R. Arreola, "Creating visuals for math instruction using JavaScript and the HTML canvas element," Ph.D. dissertation, California State University, Northridge, 2017.

[2] M. Christen, S. Nebiker, and B. Loesch, "Web-based large-scale 3D-geovisualisation using WebGL: the OpenWebGlobe project," *International Journal of 3-D Information Modeling (IJ3DIM)*, vol. 1, no. 3, pp. 16–25, 2012.

[3] R. Buels, E. Yao, C. M. Diesh, R. D. Hayes, M. Munoz-Torres, G. Helt, D. M. Goodstein, C. G. Elsik, S. E. Lewis, L. Stein *et al.*, "Jbrowse: a dynamic web platform for genome visualization and analysis," *Genome biology*, vol. 17, no. 1, p. 66, 2016.

[4] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile UI test fragility: An exploratory assessment study on Android," in *Proceedings of the 2Nd International Workshop on User Interface Test Automation*, ser. INTUITEST 2016. New York, NY, USA: ACM, 2016, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/2945404.2945406

[5] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Visual vs. DOM-Based Web Locators: An Empirical Study," in *Web Engineering*. Springer, Cham, Jul. 2014, pp. 322–340, dOI: 10.1007/978-3-319-08245-5_19. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-08245-5_19

[6] "W3C canvas elements standard." [Online]. Available: http://www.w3.org/TR/2dcontext/

[7] M. S. Tooms, *Colour Reproduction in Electronic Imaging Systems: Photography, Television, Cinematography*. John Wiley & Sons, 2016.

[8] C. Fernandez-Maloigne, *Advanced color image processing and analysis*. Springer Science & Business Media, 2012.

[9] H. Scharr, "Optimal operators in digital image processing," Ph.D. dissertation, 2000.

[10] D. Kroon, "Numerical optimization of kernel based image derivatives," *Short Paper University Twente*, 2009.

[11] M. Sezgin *et al.*, "Survey over image thresholding techniques and quantitative performance evaluation," *Journal of Electronic imaging*, vol. 13, no. 1, pp. 146–168, 2004.

[12] J. H. Sossa-Azuela *et al.*, "On the computation of the euler number of a binary object," *Pattern recognition*, vol. 29, no. 3, pp. 471–476, 1996.

[13] J. Matas, C. Galambos, and J. Kittler, "Robust detection of lines using the progressive probabilistic hough transform," *Computer Vision and Image Understanding*, vol. 78, no. 1, pp. 119–137, 2000.

[14] N. Kiryati, H. Kälviäinen, and S. Alaoutinen, "Randomized or probabilistic hough transform: unified performance evaluation," *Pattern Recognition Letters*, vol. 21, no. 13, pp. 1157–1164, 2000.

[15] M. Hadjieleftheriou, Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, "R-trees–a dynamic index structure for spatial searching," in *Encyclopedia of GIS*. Springer, 2008, pp. 993–1002.

[16] J.-B. Ryu, C.-G. Lee, and H.-H. Park, "Formula for harris corner detector," *Electronics letters*, vol. 47, no. 3, pp. 180–181, 2011.

[17] "Canvasure tool," https://github.com/msbajammal/canvasure.

[18] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[19] E. Jones, T. Oliphant, and P. Peterson, "{SciPy}: open source scientific tools for {Python}," 2014.

[20] O. Westesson, M. Skinner, and I. Holmes, "Visualizing next-generation sequencing data with jbrowse," *Briefings in Bioinformatics*, vol. 14, no. 2, p. 172, 2013. [Online]. Available: + http://dx.doi.org/10.1093/bib/bbr078

[21] D. Croft, A. F. Mundo, R. Haw, M. Milacic, J. Weiser, G. Wu, M. Caudy, P. Garapati, M. Gillespie, M. R. Kamdar, B. Jassal, S. Jupe, L. Matthews, B. May, S. Palatnik, K. Rothfels, V. Shamovsky, H. Song, M. Williams, E. Birney, H. Hermjakob, L. Stein, and P. D'Eustachio, "The reactome pathway knowledgebase," *Nucleic Acids Research*, vol. 42, no. D1, p. D472, 2014. [Online]. Available: + http://dx.doi.org/10.1093/nar/gkt1102

[22] C. A. Miller, J. Anthony, M. M. Meyer, and G. Marth, "Scribl: an HTML5 canvas-based graphics library for visualizing genomic data over the web," *Bioinformatics*, vol. 29, no. 3, p. 381, 2013. [Online]. Available: + http://dx.doi.org/10.1093/bioinformatics/bts677

[23] C. Roberts, G. Wakefield, and M. Wright, "The web browser as synthesizer and interface." in *NIME*. Citeseer, 2013, pp. 313–318.

[24] A. U. Sinha and S. A. Armstrong, "iCanPlot: Visual exploration of high-throughput omics data using interactive canvas plotting," *PLoS ONE*, vol. 7, no. 2, 2012.

[25] W. E. Winkler, "Overview of record linkage and current research directions," in *Bureau of the Census*. Citeseer, 2006.

[26] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 112–121.

[27] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[28] S. Mahajan, K. B. Gadde, A. Pasala, and W. G. J. Halfond, "Detecting and localizing visual inconsistencies in web applications," in *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, December 2016.

[29] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Automated generation of visual web tests from DOM-based web tests," in *Proceedings of the Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 775–782.

[30] B. Burg, A. J. Ko, and M. D. Ernst, "Explaining visual changes in web interfaces," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 2015, pp. 259–268.