Enabling Visual Regression Testing for Information Visualizations

Nychol Bazurto-Gomez, Mario Linares-Vásquez, John A. Guerra-Gomez

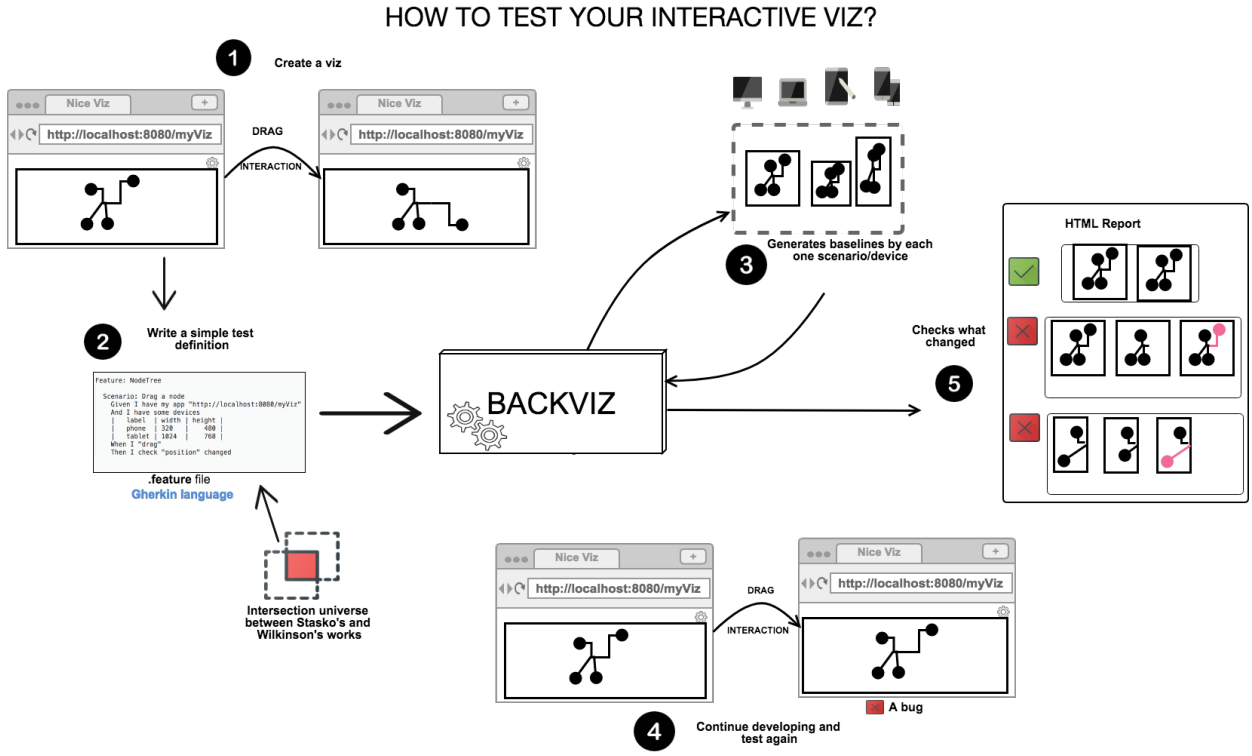


Fig. 1: BACKVIZ allows for automated testing of complex interactions on data visualizations. The image shows BACKVIZ workflow. When the visualization is running on a server, the developer can write a simple interaction test on a high level language (*i.e.,*`Gherkin`), where she can define automated tests for more complex visualization interactions such as zoom or drag. Then BACKVIZ is called to generate baselines for different devices and configurations. Afterwards, the developer can continue developing, and when she wants to check if some undesired change has been incorporated, she can just rerun the tests and collect reference snapshots. As a result, BACKVIZ will check for changes on the common marks and channels used on data visualization by means of image comparison, and will produce a report highlighting differences if any.

Abstract— Testing interactive data visualizations is a tedious and complicated task that is performed mostly manually because there are no supporting tools or techniques for automating the process. Current automated testing solutions do not allow testers to validate visual responses to complex user interactions such as drag and zoom. One potential solution to the aforementioned issues is visual regression testing, however, it forces the developer to deal with complicated libraries and hard to use languages. Moreover, it is limited to simple interactions such as click and scroll. To address this we propose BACKVIZ, a visual regression testing framework that offers three main advantages over the state of the art solutions: (i) allows developers to write tests in a high level language; (ii) enables users to test the marks and channels defined in popular visual analytics frameworks such as Munzner's [48]; and (iii) makes it easier to test complex interactions types identified in a taxonomy that intersects Wilkinson's [58] universe of Web Interactions and Stasko's [60] summary of interactions techniques. BACKVIZ is the result of a thorough analysis of the state-of-the-art techniques for software testing that reveals the need for a new category of interactive visualization testing focused on visual analytics principles.

Index Terms—Information visualizations testing, visual regression testing, framework

- *Nychol Bazurto-Gomez is with Imagine Research Group at Universidad de los Andes. E-mail: n.bazurto: uniandes.edu.co.*
- *Mario Linares-Vásquez is with The Software Design Lab at Universidad de los Andes. E-mail: m.linaresv@uniandes.edu.co.*
- *John A. Guerra-Gomez is with Imagine Research Group at Universidad de los Andes. E-mail: jaguerra@uniandes.edu.co.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

1 INTRODUCTION

Testing is considered a primordial step in software development to (i) assure the quality of both process and product, and (ii) reduce/avoid the rate of bugs that are incorporated by developers into the codebase. However, when there is no support on automation techniques, testing could be an expensive, time consuming and painful activity for software developers and testers alike. For the specific case of information visualization testing, the current lack of easy to use tools that support the fundamentals of visualization design, has left most developers testing manually for more elaborated interactions (*e.g.,* zoom and drag). Moreover, infoviz developers must test for complex changes on marks

and channels such as animations, color or shape changes. To address this, we present BACKVIZ, a framework for automated testing of information visualizations, which allows developers to test for complex interactions, using an easy to use high level language, and enabling them to test for the common marks and channels used on Visualization design, with support for multiple displays and configurations.

Before introducing BACKVIZ, let us illustrate the state of the art of GUI testing and make the argument of the need for specialized techniques for information visualization testing. Many techniques have been developed to address the specific needs of different types of applications such as mobile [29, 36–39], cloud [35], web [45, 51, 52], and big data apps [55]. There is a plethora of works devoted to specific testing approaches for each type of system, but there is also extensive previous work on general approaches, models, and theory for software testing. For instance, there is a software testing taxonomy created by the Software Engineering Institute [32] that describes all the existing types of testing (about 200 types) and organizes the types around 7 fundamental questions: *What are we testing? When are we testing? Why are we testing? Who is performing the test? Where is the testing taking place? How are we testing? How well are the objects-under-test functioning?*.

From the wide space of models, techniques, types, in the testing community, Graphical-User-Interface testing (*a.k.a.,* GUI testing) and GUI-based testing [28,42–44] have gained a lot of attention and traction in the last 20 years, because of the dominance of GUI-based software systems as part of daily life activities, including mobile apps that are widely dependent on user interaction via events on device displays (*e.g.,* tablets, phones, smart watches, TVs, etc.). Note that while GUI testing focuses on validating the quality of the GUI in software systems, GUI-based testing is in general a type of software testing that executes the system under test by interacting with the GUI.

Researchers and practitioners in both GUI testing and GUI-based testing (GUIT & GUIBT), aim at designing techniques and approaches that automatically (i) generate sequences of events that are executed on the GUI of the system under testing, (ii) validates the system behavior against testing oracles that are manually or automatically generated a-priori, and (iii) look for GUI style, GUI logic, and aesthetics issues. As a consequence, different general approaches exist for automating GUIT & GUIBT such as random input generation, model-based testing, systematic exploration (*a.k.a.,* ripping), and visual testing. We will provide provide more details in Section 2.

However, despite all the progress that has been made on GUIT & GUIBT, there is still a gap in between the testing needs and the supporting tools, in particular for Visual Regression Testing (VRT) of information visualizations. Visual Testing focuses on automated testing of GUI components across different devices size, resolutions, browsers, etc. The goal of Visual Testing is to automatically detect bugs that show up directly on the GUI and are materialized on the GUI components and layout, *e.g.,* components displayed with the wrong style or in a wrong location [34]. Visual Regression Testing, is the extension of VT but with the purpose of validating that changes to a version $x+1$ of a software system, do not inject bugs on the GUI of version x. Therefore, a *regression* is executed when test cases are designed/executed on version $x+1$ to assure that the GUI works as it was in version x.

Current approaches for VT/VRT highly rely on comparing screenshots of a baseline (*e.g.,* GUIs in version x) to screenshots from a reference version (*e.g.,* GUIs in version $x+1$) [25, 33], which is very limited to detect fine grained changes (*e.g.,* a circle visualization marker is a diamond in a new version of the system). In the particular case of Web applications, VRT can be supported on comparisons at the browser render engine level –in web browsers– with the *headless* option provided by some browsers and front end testing frameworks (*e.g.,* Headless Chrome [14], NightWatch [3], and Puppeteer [21]; the headless mode reduces the memory and CPU overhead of having a browser GUI that renders the system under testing. However a current issue is the fact that implementing VRT with existing frameworks requires of programming skills which is not a common skill in GUI designers that are used to work with visual or natural languages (there is a language dichotomy between the expressiveness of code and natural or visual language [46]).

Current VRT frameworks have not been designed to conduct VRT on information visualizations. Some interactions in information visualization systems are quite more complex than typical interactions in regular web development, so a technique as traditional VRT isn't enough. Therefore, the most common practice for testing information visualization systems is manual testing. For example, when developing an interactive network visualization, developers must test for complex interactions such as dragging a node, hovering over a cluster or zooming into a sub-network. Moreover, they need to check that when these interactions trigger, the visualization responds with changes on the marks and channels used (*i.e.,* color hue, shape, opacity, size, etc.). Furthermore, the visualization can respond to user interactions with animations, or transitions to keep the user on context of the data. These specific types of user interactions, and the required responses on the visualization, are simply not to test possible with current VRT systems, as they won't go further than testing for click or scroll. On the side of the marks and channels, some current libraries such as `Backstop` [53], support checking for visual changes by image comparison, but they require the developer to write complex scripts and to setup multiple elaborated libraries that many times to play along nicely. Thus, testing information visualizations demands tailored frameworks for VRT.

In this paper we present BACKVIZ, a novel VRT framework for information visualizations that allows practitioners to design/execute information visualization scenarios and test cases at a high level; therefore, information visualizations designers can rely on BACKVIZ to write test cases without dealing with low level programming sentences. To this, BACKVIZ draws on and extends prior research on visual regression testing, Behavior Driven Development [56], and relies on visual analytics frameworks such as Munzner's [48] and Stasko's [60] ones. In particular, BACKVIZ provides a simple way to design test cases but using the visual analytics domain vocabulary, that can be executed automatically on the system under test. The main contributions of this paper are:

- BACKVIZ, as a framework that allows testing for richer interactive features through a high-level language and with a visual analytics focus. To the best of our knowledge,this is the very first approach that combines data visualization fundamentals with software testing techniques, on an easy to use framework.
- A comprehensive universe of possible interactions for visualizations comparing of Wilkinson's [58] and Stasko's [60] frameworks.
- A prototype implementation as demonstration of BACKVIZ framework which implements three interactions from the universe.

Paper organization. Section 2 briefly describes previous work on GUIT & GUIBT, visual regression testing, and Behavior-Driven Testing (BDT). Section 3 focuses on defined Visual analytics basis, models and possible interactions. Then, BACKVIZ framework is detailed in Section 4 deepening into its design and implementation. Finally, Section 5 shows some application scenarios. Conclusions and future work close this paper.

2 AUTOMATED SOFTWARE TESTING

BACKVIZ relies on two approaches for automated testing such as VRT and BDT; thus, in this section we briefly describe the high level categories of approaches for automated GUI testing, then we describe related work for VRT, and finally we comment on Behavior-Driven Testing. Note that, this section is not pretended to be exhaustive when describing the existing software types, levels, and approaches for testing. However, we want to contextualize the reader that is not familiar with automated testing before presenting the details of our BACKVIZ approach.

2.1 Approaches for Automated Software GUI/GUI-based Testing

There are six high-level categories of approaches that have been devised and proposed to conduct automated software GUI/GUI-based testing. The difference among the approaches are mostly on the way how the sequences of interactions/events are generated, and the purpose of each approach. Those approaches are (i) random input generation, (ii) model-based testing, (iii) systematic exploration, (iv) record and replay, (v) usage of automation APIs, and (vi) visual regression testing. We describe each of the approaches as follow.

Random input generation. This category includes methods and tools that generate sequences of events/interactions on the GUI randomly from a large search space that can include invalid events (*e.g.,* click on a position that is not clickable). This type of testing aims at generating sequences capable of finding corner cases or bugs that are hard to generate by humans. There are two representative types of random generators: (i) fuzzers, and (ii) monkeys. The former focus on generating random data that can be feed trough the application GUI; the later focus on generating also random events. Well known examples of monkeys in the testing community are the `Monkey` tool for random testing of Android apps [18], and the `Gremlins` tool for random generation of inputs and events on Web apps [12]. One benefit of fuzzers and monkey is that they require human interaction only for setting the events sequence seed (if required) or for defining the groups of events to be generated in the sequence.

Model-based testing. A second approach for generating sequences of GUI events relies on models that are manually or automatically generated a-priori. A model is a collection of states and transitions (similarly to a graph), in which a state is often a window configuration in the system under test, and a transition is an event/interaction that triggers changes between the states (*i.e.,* transitions that change the GUI) [49]. In the case of models automatically derived, static or dynamic analysis are applied to analyze different artifacts such as the source code, existing diagrams, execution traces, or intermediate representations (*e.g.,* DOM models for web apps). Once the model is built, sequences of events/interactions (*i.e.,* test cases) are generated by traversing the paths in the graph. Representative examples of model-based testing are the `GUITAR` family of tools for mobile, desktop and web applications [27, 50], and the `MonkeyLab` [39] and `Sapienz` [41] tools for mobile testing.

Systematic exploration. This type of testing executes the system under test by selecting iteratively the events/interactions to be executed according to a current snapshot of the application. The goal here is to traverse as many execution paths as possible by following a predefined exploration/search strategy as deep-first or breath-first. An important component in this approach is a *ripper* that extracts a whole representation of the current GUI state and creates a model to keep track of the GUI components and events already visited/triggered. Thus, the *ripper* gets a snapshot, then the testing tool executes a valid event for the snapshots, and continues until more events can not been executed. Approaches for systematic explorations are often used with model-based testing for building models interactively. Examples from this type of testing are `Testar` [57], `DynoDroid` [40], and `Crashscope` [47]. Systematic exploration is specially used to achieve high code coverage during testing, however, it is slow because of the overhead generated when getting and analyzing the snapshots.

Record & Replay. It is probably one of the most used approaches for GUI testing/GUI-based testing. It consists on recording scripts while the system under test is manually executed. The scripts can later be reproduced automatically. Record & Replay tools allow easy collection and reproduction of scripts, however, some of the tools generate scripts that are coupled to specific locations (for the events) and window size. Therefore, reproducing scripts on windows with different dimensions might not work properly. Also, the scripts are easily deprecated when the GUI is modified drastically. Examples of tools for Record & Replay are `Selenium` [22] and `Barista` [30].

Automation APIs. The APIs allow developers and testers to write test cases (using a programming language) that can be executed on-demand or as part of a continuous integration engine. The automation APIs allows the programmer to have mechanisms for locating components in the GUI and then executing events/interactions on the components. There is a wide set of automation APIs for different types of applications, for example, `NightWatch` [3], `Cypress` [8] , `WebDriver IO` [26], and `Casper JS` [1] for web applications, and `Appium` [4], `UIAutomation` [24], and `Espresso` [9] for mobile apps.

2.2 Visual Regression Testing (VRT)

The purpose of VRT is to validate whether changes, in a new version of a system under test, do not introduce bugs into the GUI of the system. VRT was born in the web development community, and in particular in the front-end development side to assure that changes to style-sheets kept the consistency with previous versions of the GUI. However, the concepts behind VRT have recently got attention from the mobile development community to design approaches for detecting visual inconsistencies of mobile apps across different mobile platforms [31].

The general process for VRT is depicted in Fig. 2. There are two fundamental elements in VRT: the baseline, and the references. The baseline is the set of screenshots/snapshots collected on the version of the system that will serve as ground-truth for the regression, and the references are the screenshots/snapshots collected on the new versions of the system that will be compared against the baseline. Note that we use here the terms screenshots/snapshots because the comparisons during a VRT process can be done using images, or by using other representations such as hashes of the GUI, GUI hierarchies (*e.g.,* a DOM tree for a Web app), memory snapshots, among others. VRT consists on executing the same set of test cases on the baseline and the references, with the purpose of collecting evidences of inconsistencies in the GUI, when comparing the references to the baseline. As for the comparison, a *difference threshold* and similarity metrics should be defined to detect when there are inconsistencies/differences.

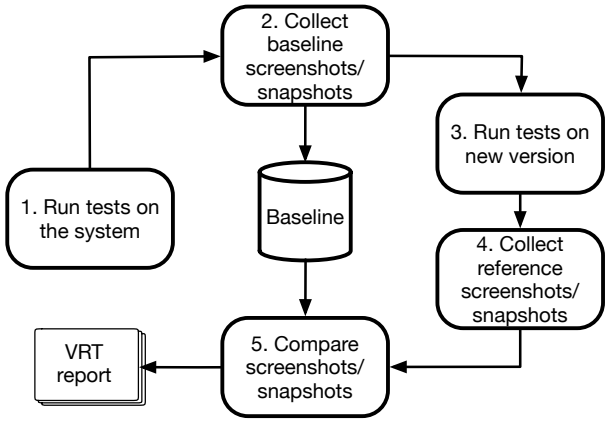


Fig. 2: Main steps for Visual Regression Testing (VRT).

While VRT can be done manually, automated VRT is desired because of the complexity inherent to conducting VRT on different combinations of browser, device, resolution, operations system, etc. Several tools are publicly available for supporting VRT in web applications such as `Spectre` [23], `PhantomCSS` [20], `CSS Critic` [7], `Ocular JS` [19], `Backstop JS` [53], `Galen` [10], and `Hardy` [13]. The aforementioned tools allow for (i) execution of test cases, (ii) collection of screenshots/snapshots for the baseline and the references, and (iii) comparison of screenshots/snapshots. However, the tools were not designed to support VRT on information visualizations because data visualizers and visual analytics systems are designed to support complex events/interactions not supported by state-of-the-art VRT tools. Moreover, most of the available tools require of programming skills from the users, because the test cases to be executed and the steps in the VRT process need to be programmed using languages such as JavaScript, Ruby, or Python. Note that this is a limitation for designers of information visualization systems that are used to work with examples and tests cases written visual or natural languages.

2.3 Behavior-Driven Development (BDD) and Behavior-Driven Testing (BDT)

BDD is a software development methodology that is built upon the best practices of specification by example and Test-Driven Development [56, 59]. BDD tries to reduce the communication issues between the domain experts and the programmers by relying on a ubiquitous language that is used to (i) "write acceptance tests as examples that anyone on the team can read" [59], and (ii) write specifications that can be executed directly on the system and operate like live documentation. An scenario/acceptance test is mainly composed of individual steps that describe the events/interactions to be executed on the system, and the acceptance criteria for the scenario. In BDD, both, domain experts and programmers, specify usage scenarios in a high level-language (*e.g.,* Gherkin); then, the scenarios can be executed directly on the system as acceptance tests, by relying on a testing framework that allows for defining low-level steps programmatically or (*i.e.,* the code of the test that is executed on the system) reusing pre-defined events/interactions[1]. Some of the available frameworks for BDD are `Cucumber` [2], `Jasmine` [15], `Calabash` [5], `JBehave` [16], `LightBDD` [17], `Chakram` [6], and `Gauge` [11].

We briefly present BDD here because we are interested on including into VRT, only the testing related practices from BDD, *i.e.,* we are interesting only on the Behavior-Driven Testing (BDT) part. In particular, our goal with enabling VRT for information visualizations is to provide designers of visualization systems with an easy way to specify usage scenarios for visualization systems, by using an ubiquitous language and terms from the information visualizations domain. Given the need for continuous deployment of visualization systems, automated testing (and regression testing) of information visualizations is a must. Testing processes for information visualizations should be agile enough to be executed in a continuous integration fashion with little human intervention and support modern technology stacks (*e.g.,* D3, JavaScript, NodeJs). Consequently, we see BDT as a promise for VRT of information visualizations; BDT should allow easy generation of acceptance test cases (for information visualization systems) that reuse steps definitions from a pre-defined library of events/interactions.

None of the current approaches and tools for GUI/GUI-based testing, VRT and BDD have been designed for information visualization systems. On the one side, there is a conceptual gap between domain language used in information visualizations and expressiveness of code language used by automated testing frameworks. On the other side, current frameworks only support basic events (e.g., click, double-click, long-click, swipe/scroll, text input) that are far from being representative of complex interactions in information visualization systems.

3 Interactions and Models for Information Visualization

BackViz is also based on two visual analytics foundations: a framework which standardizes data visualization design and another one which defines the interactions universe. The latter is an intersection and categorization between two interactions analysis made by Wilkinson [58] and Stasko [60].

3.1 Tamara Munzner's framework

Design and definition of visualizations is an art, and as one, every data visualizer could define in different ways each element/interaction depending on what she wants to communicate and what user tasks she wants to enable . However, Munzner [48] detected that no matter how diverse could be a domain and its data, a visualization is independent of the domain. So, why not to reuse some design and definitions for those visualizations which looks for similar purposes. Thus, to speak the same language, Munzner established a framework to analyze visualizations according to three questions: (i) **What** data the user sees? (ii) **Why** does the user want to use a visualization tool? and (iii) **How** are the visualizations constructed in terms of design choices? We describe her answers to these three questions as follow:

What: Data abstraction. This part is the central pillar of the framework because a visualization design is strongly impacted by the data. Some extra questions related to data abstraction arise here: what is the data expressing? What information do you want to be explicit or implicit in a visualization? For example, if data is mainly about relationships between developers in Github or talk about how some countries and its primary activities are related to its geographical position, it could be proper to abstract the data as a network or geometry, even just like a table. Table 1 lists the abstract types of what can be visualized. These ones are made up of different combinations of the five data types: items, attributes, links, positions, and grids.

Table 1: Munzner's data definition [48].

Dataset	Data types
Tables	*Items *Attributes
Networks and trees	*Items (nodes) *Links *Attributes
Fields	*Grids *Positions *Attributes
Geometry	*Items *Positions
Clusters Sets, Lists	*Items

Why: Tasks. The second step after abstracting the data consists of thinking about what are the main tasks to cover/achieve with the visualization. This part allows a standard definition to identify different domains with similarities. Munzner uses action-target pairs that describe the task and the data related to the task (See Fig. 3) . The targets can be trends, outliers, dependency, distribution, among others. Figure 3 shows all targets and definitions to analyze visualizations according to Munzner's framework.

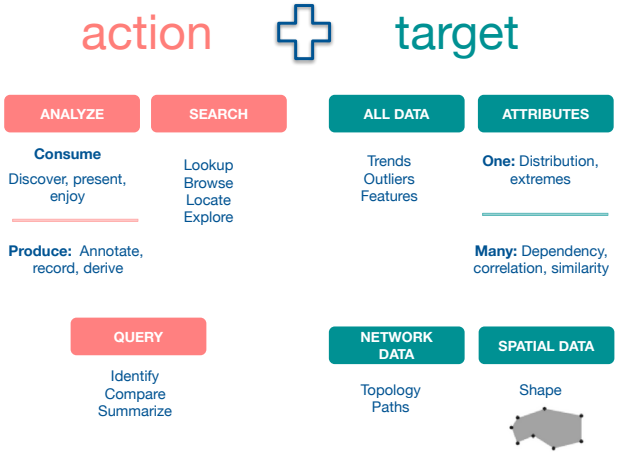


Fig. 3: Actions and targets defined by Munzner.

How: Encoding. Once the tasks were defined and prioritized, the next and last step focuses on how a visualization idiom can be constructed out of a set of design choices. Table 2 provides a list of encoding options and design options introduced by Munzner. Also, in this step it is necessary to keep in mind mark and channels, as the basis to analyze visual encoding: marks are primitive geometric objects in visualizations such as points, lines, and areas; channels are the marks appearance attributes color, position, shape, tilt, and size.

3.2 Interactions universe: Classification based on Stasko's taxonomy and Wilkison's web interactions

Keeping in mind the encoding stage from Munzner's framework, activities like select or filter appear as a design choice. That kind of

[1]Note that in this section we will not give all the details behind the BDD methodology because we are interested only on the specification by example and automated acceptance testing aspects, however, we refer the interested reader to the Wynne and Hellesoy book [59].

Table 2: Design choices defined by Munzner [48].

How choice	Types
Encode	**Arrange**: • Express • Separate • Orden • Align • Use **Map**: • Color • Size, Angle, Curvature. • Shape • Motion
Manipulate	• Change • Select • Navigate
Facet	• Juxtapose • Partition • Superimpose
Reduce	• Filter • Aggregate • Embed

decisions could be considered as interactions. BACKVIZ is addressed to test interactions and its behavior along the coding time, so, it was necessary to look for interactions taxonomies to establish reference set to follow. Notably, there are previous surveys focused on interactions as fundamental part of information visualization, highlighting Stasko's work [60] for its completeness. It summarizes all relevant taxonomies to interaction techniques and its taxonomic units. An example of the kind of taxonomies considered is [54] which raises its mantra, including the interactions: overview, zoom, filter, details-on-demand, relate, history, and extract. Additionally, Stasko established seven categories that consider all kind of interaction techniques and try to describe the essence of **the user's intent**.

However, we needed an explicit list of interactions. Therefore, we turned our attention to Wilkinson's work [58], which it was considered in Stasko's literate review as a taxonomy of low-level interaction techniques. Wilkinson's work mentions an entire set of possible interactions for Web-based systems. Both works, Stasko's and Wilkinson's are essential for us, which results in an intersection and a new classification of those works that allowed us to understand better all the interactions spectrum. Figure 4 shows that categorization.

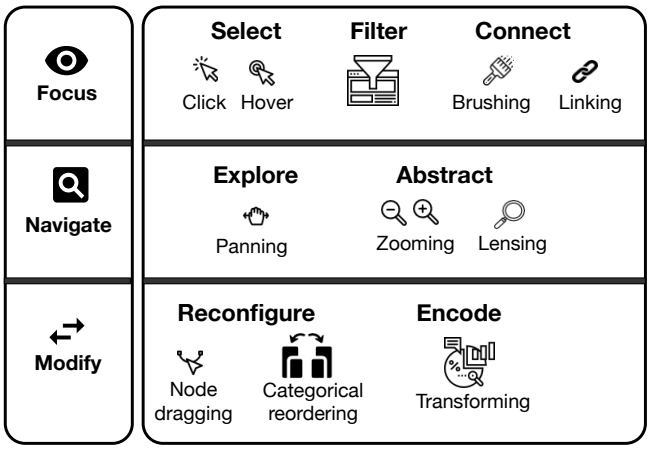


Fig. 4: Categorization based on Stasko's and Wilkinson's works.

The first layer in Fig. 4 (left) indicates a high-level categorization that we defined for Stasko's categories:

- **Focus:** this category groups *select*, *filter* and *connect*, which main characteristic is to allow the user tracking some important information,*e.g.,* highlighting associations and relationships.
- **Navigate:** as its name indicates, it wraps those Stasko's categories that enable get details through navigation-related interactions. *e.g., Explore* and *abstract*.
- **Modify:** refers to interactions that offer a different perspective of the data, such as *Reconfigure* and *Encode*.

The second layer in Fig. 4 (right) contains Stasko's categories and inside each one of these are Wilkinson's interactions. Each category and Wilkinson's interactions that we classified inside them, are detailed as follows:

- **Select:** provide users with the ability to **mark** something as **interesting**. This kind of interaction is useful when the number of items is large, making it visually distinctive. *Interactions: Click, Hover*
- **Explore:** let the users **examine** a different subset of data. Usually, an InfoVis system shows an overview of data at a time due to screen and perceptual and cognitive user limitations, thus, examining a subset of data improves data understanding. *Interactions: Panning*
- **Reconfigure:** related to showing a different arrangement of the data set, looking for hidden characteristics like relationships. *Interactions: Node dragging, categorical reordering*
- **Encode:** enable users to see a different representation, altering the fundamental one. It includes appearance changes in color and shape; these can affect how users understand relationships and distributions. *Interactions: Transforming*
- **Abstract:** as part of the famous visual analytics Shneiderman's Mantra [54], details on demand is vital, and this interaction technique is related to that. Therefore, "abstract" allow users to go from an overview down to details of individual data, like in a GUI component tooltip *Interactions: zooming, lensing.*
- **Filter:** it means showing something in a restrictive way but based on some specific criteria. *Interactions: Categorical filtering, continuous filtering, multiple filters, fast filtering.*
- **Connect:** as its name indicates, the main idea here is to show related items, either highlighting associations between items or showing/ hiding items that are important to another one. *Interactions: brushing, linking*

The mentioned interactions are complex and in conjunction with the constant use of Munzner's framework concepts, they stand out the need for a specific VRT framework that allows testing of complex interactions but while keeping the domain terms used in the aforementioned visualization frameworks. Current tools do not address these needs and implementing testing solutions with current tools requires a lot of extra effort from developers to tailor the available tools.

4 BACKVIZ: A NEW FRAMEWORK FOR VRT OF INFORMATION VISUALIZATIONS

BACKVIZ focuses on making easier to conduct regression testing of data visualizations. In particular, BACKVIZ allows developers, testers, and information visualizations designers to (i) write test cases using specifications based on visual analytics concepts (*i.e.,* marks and channels, and its interactions); (ii) use visual regression testing as the underlying technique to detect visualization changes/inconsistencies across different versions of the system; and (iii) extend the universe of predefined interactions on-demand.

4.1 BackViz design

Figure 1 illustrates entire flow for testing a visualization using BACKVIZ. The first step shows a network visualization which has a drag interaction. This visualization is running on a server which is necessary. Then, the developer must write his tests with the regarding structure, using keywords like `when` and `then`. Those tests are going to call BACKVIZ engine which includes corresponding step definitions and modules to support the visual regression testing. An important characteristic is that developer can specify the devices that he wants BACKVIZ supports. To initialize all the process is required to establish a baseline,

for that purpose a command must be executed. This command it is going to generate a first pair of images with the behavior of the drag interaction. Once the baseline has been taken, developer can continue coding his visualization. When he wants to test his work must run his tests files by a command. This is going to trigger a process which consist on:

- **Evaluate the step definitions**: Defined scenarios are going to be mapped to the corresponding step definitions. These definitions identify the used vocabulary, in the case of Figure 1, it is going to detect that needs to send an instruction to execute a drag interaction over the running application.

- **Call VRT engine**: Once an entire scenario is read, VRT module is called. This module is going to use a Headless Browser to evaluate the interactions emulating each devices that the developer defined. A new image is taken from result to execute the interaction. After that, an image comparison process is made between the baseline image(s) and those ones have been taken recently. It applies a standard configuration if nothing related to accuracy it was defined in the test file. This accuracy parameter can affect how perceives a change depending on how subtle it is. For the specification in Figure 1 , it is going to look for differences in position (drag should affect that channel). But it also can detect differences in color.

- **Generate a report**: When image comparison process finalizes, a report is presented. This document allows developer to identify on a graphic way ,if his new code lines have affected the visualization and its interactions.

The last steps are going to be executed each time that developer runs the test file. Now, if the developer needs add an extra scenario he should run the baseline process again. These is due the baseline is based on a preliminary test file and just took images about that interactions.

Finally, BACKVIZ works with some constrains:

- Developer must test with the same data that application was running when the baseline was taken. Due the image comparison looks for difference in position and color, probably different data are going to throw a 100% mismatch.

- If a scenario was defined for some devices and needs to run over another ones, it is going to need a new baseline or test file. BACKVIZ was thought to work again the baseline and new characteristics are going to produce a mismatch.

- Test file should be define using a properly structure and allowed descriptions.

4.2 BackViz implementation

BACKVIZ prototype relies on some specifics technologies as Figure 5 shows. It is composed of three main components: (i) the BDT module, (ii) the Bridge Engine that executes the BDT instructions and create a properly JSON file to trigger the visual regression testing process based on it, and (iii) the VRT module which uses `Backstop.js` technology. We decided to use `Backstop.js` instead another technologies because most of traditional VRT tools allows the same interactions set, even private ones.

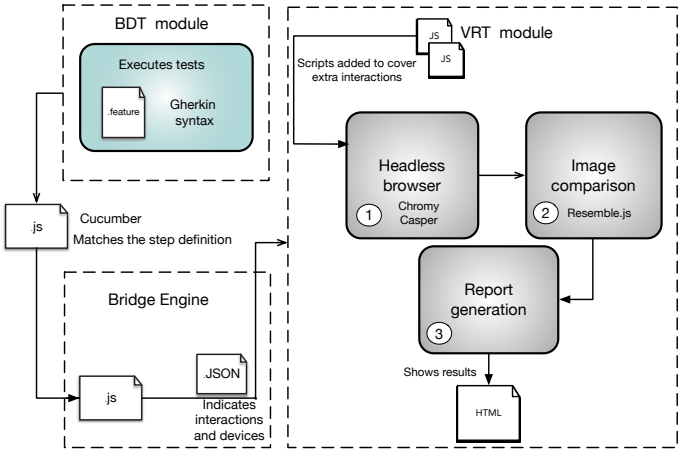


Fig. 5: BACKVIZ architecture.

4.2.1 The BDT module

Data visualizers, regular developers or testing enthusiasts can use BACKVIZ features by the Behavior Driven module due its simplicity. It provides a easy way to write the scenarios to be tested, using natural-language-based descriptions. This module works thanks to `Gherkin` and the underlying technology of `Cucumber.io`. A test file should have `.feature` extension and contains specifications written in `Gherkin` language. `Gherkin` has keywords that are going to be used to starts each line. After those keywords can be any text aligned with the BACKVIZ's step definitions. As it is possible see in the next snippet, keywords are `Feature`, `Scenario`, `Given`, `And`, `When` and `Then`.

```
Feature: Barchart

  Scenario: Hover a bar
    Given I have my app "http://localhost:7000/"
    And I have ".bar" class
    And I have some devices
    |   label  | width | height |
    |   phone  | 320   |    480 |
    |   tablet | 1024  |    768 |
    When I "hover"
    Then I check "color"
```

But these scenarios aren't enough by themselves, Cucumber needs *step definitions* to execute them. So, BACKVIZ define a set of step definitions per each allowed step in a Scenario. A step definition is a `.js` file with it defines patterns that are going to be linked with the scenarios. That code is what Cucumber calls when read a Gherkin definition written by the developer.

4.2.2 The Bridge engine

The Bridge Engine (BE) supports the communication between the other two BACKVIZ's modules. Once an entire scenario is read, a script is called to generate the proper configuration for the Visual Regression Module. This configuration is a JSON and is defined in terms of Backstop needs. So, interactions required by the user become in an particular pair-value, for instance:

```
...
      "hoverSelector": "",
      "clickSelector": "",
      "dragSelector": "true",
...
```

4.2.3 The VRT module

`Backstop.js` as was mentioned, is a Visual Regression Testing tool. It is founded on two main technologies: (i) Resemble.js to the image comparison, and (ii) the headless browser that could be `Chromy` or `Casper.js`. It necessary makes clear that `Backstop.js` does not

have a huge interactions set to test. In fact, just define click interaction and hover for Chromy and Casper (with some bugs). So, to use another interaction BACKVIZ prototype defined extra scripts inside `Backstop.js` .

5 BACKVIZ PROTOTYPE ON ACTION

To demonstrate how BACKVIZ works, two usage scenarios are described in this sections: (i) a basic bar chart visualization with a hover interaction, and (ii) a scatterplot with changes in the marks and channels[2]. However, first let us summarize how VRT would be done in but without the support of BACKVIZ. A typical scenario with the existing VRT tools, requires from the tester/developer to:

- Understand how the tool reads the steps/scenarios. For BackstopJs, the developer should look for the file Backstop.json and start to modify to define the desired scenarios.
- Identify the Headless Browser scripts and include the need interactions. If the user hasn't worked on that, it adds time to learn syntax and how works. Usually, VRT tools come with click interaction.
- Understand the image comparison system to enable/disable a factor.

These steps narrow the potential user profiles. They are reduced to developers with good coding skills who could implement, in a prudential time, the flow that they need. Moreover, to share the tests, and that knowledge incurs in extra time.

On the other hand, it is always possible to test in a manual fashion, but it is evident that it is not efficient. Running a viz and do as much interaction as possible over every component is time consuming and could leave many bugs behind.

5.1 Barchart scenario

Suppose you are developing a small visualization, one with a bar chart. Your first attemp doesn't have interactions, and the bars are red. You want to know what happens if you take a BACKVIZ baseline and run a simple test. For that purpose you write a test case as the following:

```
Feature: First barchart
  Scenario: Nothing changes
    Given I have my app "http://localhost:7000/"
    And I have ".bar" class
    And I have some devices
    |   label   | width | height |
    |   phone   | 320   |    480 |
    Then I check "color"
```

Because you don't implemented any changes since the baseline was collected, BACKVIZ is going to answer to you with no errors/no differences, same data, same color, as illustrated in Figure 6.

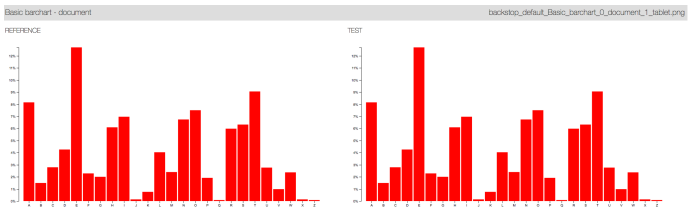


Fig. 6: BACKVIZ analyzes a visualization that it didn't change.

Now, you changed your visualization color to blue . After that, you remember to run the command to retake the baseline (now BACKVIZ is going to compares against a blue bar chart). Later, you decide to add a hover interaction, so you include it, and a bar will become green when it happens. What occurs if you change the scenario file to include the evaluation of the hover interaction? To make that you must specify in your test case the hover interaction:

```
...
    When I "hover"
    Then I check "color"
...
```

Your new interaction is considered as an error/difference as Figure 7 shows. This event results from maintaining the baseline that was evaluated without interaction (just the blue change). The same variation will appear if you make changes in your code and delete/bug that interaction and this one is in the baseline.

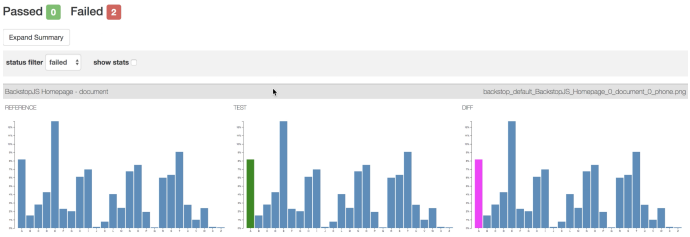
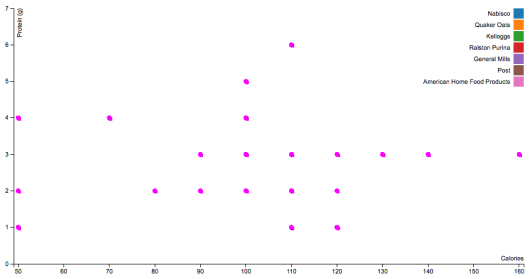
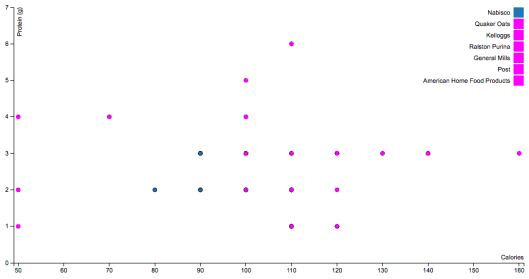


Fig. 7: BACKVIZ reports a mismatching due to the new interaction it wasn't evaluated in the baseline.



(a) Result when dots were changed by square (shape change).



(b) Result when color scale change and affects some values.

Fig. 8: Results from scatterplot scenario when some bugs/changes were incorporated.

5.2 Scatterplot scenario

Other type of changes that are possible in information visualizations are related to the marks. Additionally, a color scale different could generate partial changes that are hard to detect depending on the accuracy settings of the image comparison process. In the scatterplot case, it is necessary to defined a threshold value increase to detect all the color/shape changes. The main reason for this is that a point has a smaller affected area.

The scatterplot in Fig. 8 handles points as mark, so, the same shape represents all the values. If those points are switched by squares, BACKVIZ will identify those differences, highlighting each point position

[2]Videos showing BACKVIZ in action are available with our online appendix `http://nycholbazurto.me/BackViz/`

(Figure 8 (a)). If the color scale changes, less/more colors can represent all the data. Some points could preserve the same hue, but some categories are going to be affected definitely. BACKVIZ will show differences depending on resemble.js thresholds. For Figure 8 (b) it is possible to appreciate that blue color was the only one which didn't change.

6 CONCLUSION AND FUTURE WORK

This work presents the design and implementation of a prototype framework to perform visual regression testing on visualizations: BACKVIZ. From a study and analysis of the possibly related techniques and with a set of tools from the testing area, a vertical technology is constructed that allows writing tests with a visual analytics emphasis. Our design focused on supporting the data visualization basis such as marks, channels and a diverse set of interactions.

The main contributions of this work are the design of a framework and its corresponding concept proof which serves as a first approximation to cover the lack of a VRT technique for such purposes. It supports an essential process for visualizers, letting them establish their tests and applied along the time, detecting bugs or undesired changes incorporated through new releases.

In BACKVIZ, the whole technological flow is established, based on visual regression testing. However, other possible testing techniques could be applied as would be static code analysis, to narrow the visualizer scenarios and guarantee that their tests are defined explicitly within what is possible according to the code. For the latter, it is proposed to carry out studies to determine the viability to add this type of analysis, considering the integration of the tools that support them in the reference architecture model proposed in this paper. Also, future work will be devoted to explore different techniques which supports interactions with different datasets.

ACKNOWLEDGMENTS

We would like to thank Jean Daniel Fekete, Scientific Leader of the INRIA, Jose Tiberio Hernandez, and David Álvarez, both professors at Universidad de Los Andes for their valuable feedback.

REFERENCES

[1] Casperjs: Navigation scripting & testing for phantomjs and slimerjs. http://casperjs.org/, 2011.
[2] Cucumber: Simple, human collaboration. https://cucumber.io/, 2015.
[3] Nightwatch.js: Browser automated testing done easy. http://nightwatchjs.org/, 2015.
[4] Appium: Automation for apps. http://appium.io/, 2018.
[5] Calabash: Automated acceptance testing for mobile apps. https://calaba.sh/, 2018.
[6] Chakram. http://dareid.github.io/chakram/, 2018.
[7] Css critic. http://cburgmer.github.io/csscritic/, 2018.
[8] Cypress: Automation for apps. https://www.cypress.io/, 2018.
[9] Espresso. https://developer.android.com/training/testing/espresso/index.html, 2018.
[10] Galen. http://galenframework.com/, 2018.
[11] Gauge. https://gauge.org/, 2018.
[12] Gremlins. https://github.com/marmelab/gremlins.js/blob/master/README.md, 2018.
[13] Hardy. https://github.com/thingsinjars/Hardy, 2018.
[14] Headless chromium. https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md, 2018.
[15] Jasmine: Behavior-driven javascript. https://jasmine.github.io/, 2018.
[16] Jbehave. http://jbehave.org/, 2018.
[17] Lightbdd: Lightweight behavior driven development test framework. https://github.com/LightBDD/LightBDD, 2018.
[18] Monkey. https://developer.android.com/studio/test/monkey.html, 2018.
[19] Ocular. https://github.com/vinsguru/ocular, 2018.
[20] Phantom. http://phantomjs.org/, 2018.
[21] Puppeteer. https://github.com/GoogleChrome/puppeteer, 2018.
[22] Selenium, 2018.
[23] Spectre. https://github.com/wearefriday/spectre, 2018.
[24] Uiautomation. https://developer.android.com/reference/android/app/UiAutomation.html, 2018.
[25] Visual regression testing. https://visualregressiontesting.com/articles.html, 2018.
[26] Webdriver io. http://webdriver.io/, 2018.
[27] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software*, 32(5):53–59, sep 2015. doi: 10.1109/MS.2014.55
[28] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679 – 1694, 2013. doi: 10.1016/j.infsof.2013.03.004
[29] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pp. 429–440. IEEE Computer Society, Washington, DC, USA, 2015. doi: 10.1109/ASE.2015.89
[30] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. Barista: A technique for recording, encoding, and running platform independent android tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 149–160, March 2017. doi: 10.1109/ICST.2017.21
[31] M. Fazzini and A. Orso. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pp. 308–318. IEEE Press, Piscataway, NJ, USA, 2017.
[32] D. Firesmith. A taxonomy of testing types, 2015.
[33] M. Godbolt. *Frontend Architecture for Design Systems: A Modern Blueprint for Scalable and Sustainable Websites*. Oreilly Media, Incorporated, 2016.
[34] A. Issa, J. Sillito, and V. Garousi. Visual testing of graphical user interfaces: An exploratory study towards systematic definitions and approaches. In *2012 14th IEEE International Symposium on Web Systems Evolution (WSE)*, pp. 11–15, Sept 2012. doi: 10.1109/WSE.2012.6320526
[35] W. Jun and F. Meng. Software Testing Based on Cloud Computing. In *2011 International Conference on Internet Computing and Information Services*, pp. 176–178. IEEE, sep 2011. doi: 10.1109/ICICIS.2011.51
[36] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, April 2015. doi: 10.1109/ICST.2015.7102609
[37] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 613–622, Sept 2017. doi: 10.1109/ICSME.2017.47
[38] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 399–410, Sept 2017. doi: 10.1109/ICSME.2017.27
[39] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pp. 111–122. IEEE Press, Piscataway, NJ, USA, 2015.
[40] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 224–234. ACM, New York, NY, USA, 2013. doi: 10.1145/2491411.2491450
[41] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pp. 94–105. ACM, New York, NY, USA, 2016. doi: 10.1145/2931037.2931054
[42] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins. The first decade of GUI ripping: Extensions, applications, and broader impacts. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 11–20. IEEE, oct 2013. doi: 10.1109/WCRE.2013.6671275
[43] A. Memon, A. Nagarajan, and Q. Xie. Automating Regression Testing for Evolving GUI Software *.
[44] A. M. Memon, M. L. Soffa, M. E. Pollack, A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. *ACM SIGSOFT Software Engineering Notes*, 26(5):256, sep 2001. doi: 10.1145/503271.503244

[45] A. Mesbah. Chapter five - advances in testing javascript-based web applications. vol. 97 of *Advances in Computers*, pp. 201 – 235. Elsevier, 2015. doi: 10.1016/bs.adcom.2014.12.003

[46] K. Moran, C. Bernal-Cárdenas, M. Linares-Vásquez, and D. Poshyvanyk. Overcoming language dichotomies: Toward effective program comprehension for mobile app development. In *IEEE/ACM International Conference on Program Comprehension 2018*, p. to appear, 2018.

[47] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 33–44, April 2016. doi: 10.1109/ICST.2016.34

[48] T. Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series. CRC Press, 2014.

[49] B. N. Nguyen and A. M. Memon. An observe-model-exercise paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering*, 40(3):216–234, March 2014. doi: 10.1109/TSE.2014.2300857

[50] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, Mar 2014. doi: 10.1007/s10515-013-0128-9

[51] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A study of causes and consequences of client-side javascript bugs. *IEEE Transactions on Software Engineering*, 43(2):128–144, Feb 2017. doi: 10.1109/TSE.2016.2586066

[52] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah. Detecting unknown inconsistencies in web applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pp. 566–577. IEEE Press, Piscataway, NJ, USA, 2017.

[53] G. Shipon. Backstopjs 3: Visual regression testing for web apps., 2015.

[54] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pp. 336–343, Sep 1996. doi: 10.1109/VL.1996.545307

[55] H. M. Sneed and K. Erdoes. Testing big data (Assuring the quality of large databases). In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–6. IEEE, apr 2015. doi: 10.1109/ICSTW.2015.7107424

[56] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 383–387, Aug 2011. doi: 10.1109/SEAA.2011.76

[57] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener. Testar: Tool support for test automation at the user interface level. *Int. J. Inf. Syst. Model. Des.*, 6(3):46–83, July 2015. doi: 10.4018/IJISMD.2015070103

[58] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[59] M. Wynne and A. Hellesoy. *The Cucumber Book. Behavior-Driven Testing for Testers and Developers*. The Pragamatic Programmer, 2012.

[60] J. S. Yi, Y. a. Kang, and J. Stasko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, Nov 2007. doi: 10.1109/TVCG.2007.70515