Automated Visual Analysis of Non-Functional Web App Properties

by

Mohammad Bajammal

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

February 2022

Abstract

Non-functional software properties capture qualitative and generic aspects about software. Such aspects are often high level and more semantic compared to the more precise and quantitative functional properties or requirements, and therefore have been more difficult to analyze and automate. A scarcely explored, and potentially useful, alternative paradigm is the adoption of what might be referred to as a visual analysis approach to software engineering, which involves extracting or analyzing visual information pertaining to the software, with the objective of addressing software engineering problems.

The goal of the work presented in this dissertation is to improve non-functional web UI properties using automated visual analysis. We focus on particular problems of testability, accessibility, and maintainability because they have not been amenable to automation so far. First, we improve testability by converting the inherently non-testable web canvas elements into testable ones. The automated technique is based on visually analyzing the structure and properties of the canvas contents, then augmenting them into the canvas element to make it testable. Then, we propose an approach to test semantic accessibility. It is based on visually analyzing various regions of the page and then inferring any associated semantic roles, after which the UI markup is examined to assert the presence of the roles. Next, we introduce an automated technique for addressing the common problem of inaccessible web form labeling. It is based on constructing visual cues from the form, then solving for the optimal labeling associations, which are finally augmented into the inaccessible web forms to make them accessible. Finally, we present a UI component generation technique to improve maintainability. The technique first detects visual patterns in the UI, then combines subsets of these patterns into a shared template, which is finally formulated as a UI component. Our evaluations show that the proposed techniques are able to carry the inferences, analyses, and tests in an accurate and effective manner.

Lay Summary

When engineers analyze or test qualitative aspects of software, such as how accessible is the software, they often have to do so manually because there are not many techniques that can automatically help them in their testing. The work in this dissertation provides a solution to such cases. We propose techniques to analyze visual information about web apps in order to obtain useful and high level information about the app. Using this analysis approach, the dissertation demonstrates how can we make web apps easier to test, made more accessible for users with disabilities, and made easier to maintain.

Preface

Each chapter of this dissertation has a corresponding research paper. I have collaborated with my supervisor, Prof. Ali Mesbah, for conducting the research projects in all chapters. I was the first author and main contributor for all papers, including formulating the idea, design and development of the approach, and evaluation of the work. I had the collaboration of Andrea Stocco in Chapter 2 and Davood Mazinanian in Chapters 2 and 6, both of whom are former post-doctoral fellows in our lab. The papers for each chapter are as follows:

- Chapter 2:
  - A survey on the use of computer vision to improve software engineering tasks [33], repo: [255]. Mohammad Bajammal, Andrea Stocco, Davood Mazinanian, and Ali Mesbah. Accepted in IEEE Transactions on Software Engineering (TSE) 2020.
- Chapter 3:
  - Web canvas testing through visual inference [30], repo: [213]. Mohammad Bajammal and Ali Mesbah. Proceedings of the International

Conference on Software Testing, Verification and Validation (ICST) 2018. Distinguished paper award.

- Chapter 4:
  - Semantic Web Accessibility Testing via Hierarchical Visual Analysis [31], repo: [120]. Mohammad Bajammal and Ali Mesbah. Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE) 2021.
- Chapter 5:
  - Automated web form accessibility labeling using visual cues. Mohammad Bajammal and Ali Mesbah. Under submission to an upcoming software engineering venue.
- Chapter 6:
  - Generating reusable web components from mockups [32], repo: [23].. Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) 2018.

In addition, I have worked on the following projects but they were not included in this dissertation as they are outside the scope of the dissertation.
- Style-Guided Web Application Exploration. Davood Mazinanian, Mohammad Bajammal, Ali Mesbah. arXiv:2111.12184 (2021).
- Page Segmentation using Visual Adjacency Analysis. Mohammad Bajammal, Ali Mesbah. arXiv:2112.11975 (2021).

Table of Contents

List of Tables

List of Figures

Acknowledgments

First and foremost, I wish to express my deepest gratitude to my supervisor, Prof. Ali Mesbah. His guidance and expertise were instrumental throughout the entire duration of my doctoral studies, and his insight and support has made the work in this dissertation possible. My sincere gratitude also goes to my committee members, Prof. Julia Rubin, Prof. Sathish Gopalakrishnan, and Prof. Philippe Kruchten.

I would also like to thank all my dear friends and colleagues in our research lab, Amir, Andrea, Davood, Mifta, Marjana, Nashid, Quinn, Rahul, Saba, and Sijia for our lively discussions. To my parents, Faiga and Salem, I am eternally grateful for your endless love, understanding, and support throughout the years.

Chapter 1

Introduction

Web applications are typically composed of two main parts: the front-end and the back-end. The front-end encapsulates all the user facing elements of the application. It includes aspects such as the user interface (UI) layout, design themes and styles, and handling user interaction. The back-end contains the remainder of the application and includes functionalities such as database access and server setup and operation.

The development of front-end or UI for web apps is often a manual and time consuming task. In a survey of more than 5,700 web app developers, 51% reported working on front-end development tasks on a daily basis [121], more so than all other development tasks (e.g., databases, back-end integration). Another study also showed that an average of 48% of the code size of software is dedicated to the user interface [190]. Accordingly, there is a need to reduce the manual effort and time spent on the development, testing, and analysis of UI.

UIs can have both *functional* and *non-functional* properties or requirements. Non-functional properties are generic and qualitative aspects of the software [40, 68], whereas the functional properties or requirements capture specific and precise behavior. Non-functional properties describe general high-level qualities of the software that do not depend on a single specific function or element of the software, whereas the functional requirements or properties are fine-grained and exact. For instance, a functional requirement would describe what should happen exactly when a *specific* button on the UI is pressed, thereby defining the function of that particular button. In contrast, a non-functional property such as accessibility applies to the UI in general, such as requiring that any part of the UI is coded using markup that expresses its high-level purpose (e.g., navigation areas, menus) whenever meaningful. These are different from the functional requirements which specify, for instance, that clicking a specific button x shows a menu y. The non-functional properties are therefore more qualitative and high-level compared to the more precise and exact functional ones.

Most areas of the software engineering lifecycle (e.g., requirements, development, testing) often have the ultimate goal of contributing to a fundamental product of software engineering: the source code. Accordingly, a wide range of software research and development activities have typically revolved around the source code, whether to improve its quality and reliability, or increase developers' productivity. As such, most existing techniques that analyze or test web UI tend to be code-based analyses, which focus on examining the source code of the app.

A scarcely explored, and potentially useful, recent alternative paradigm is the adoption of what might be referred to as a *visual analysis* approach to software engineering [33]. This approach involves the use of algorithms that extract or analyze any visual aspect pertaining to the software. The objective is to address software engineering tasks or problems by utilizing a complementary (i.e., visual) perspective of the software instead of relying exclusively on the code. For instance, a typical visual analysis approach might involve visually comparing a pair of screenshot images of different versions of a software's UI for testing purposes, such as testing for regressions or for cross-browser incompatibilities. In Chapter 2, we conduct an in-depth investigation in terms of exploring and classifying visual analysis techniques and how they are used.

1.1 Challenges and Motivation

The goal of this dissertation is to investigate how visual approaches can automate the analysis and testing of non-functional web UI properties. Non-functional properties have received less research attention compared to functional ones [40, 126]. The qualitative nature of non-functional properties makes them harder to precisely analyze and test. Their qualitative nature has also limited the extent to which they can be properly automated since many aspects often require a high-level qualitative judgment by a human observer [8, 25]. For instance, accessibility is a non-functional property or requirement that expects a web UI to be implemented using markup code that reflects its *intended purpose* [1]. That is, if some portion or region of web UI is meant to be the main section of the page as opposed to being, for instance, a navigation region or footer of the page, then the code of the UI must reflect that specific intended purpose. Inference and understanding of the possible intended purposes within a UI requires a qualitative and semantic judgment that no existing approaches address [31, 256]. Accordingly, the testing and analysis of non-functional properties has typically remained a manual and laborious time consuming process.

In this dissertation, we focus on the particular problems of the non-functional properties of testability, accessibility, and maintainability because they have not been amenable to automation so far, as will be discussed in the next sections. In addition, these properties also sample various stages of the software engineering lifecycle (i.e., development, testing, maintenance). Given that web UI is visual in nature, we believe that it would be a suitable fit to use a visual analysis approach to analyze various aspects of web UI. This can be especially advantageous in advancing the state-of-the-art of analyzing non-functional UI properties due to their high-level qualitative nature. In other words, as will be demonstrated in the following subsections, a visual analysis approach might be able to capture and imitate some of the high-level visual understanding that is currently being done manually by developers, and therefore might be able to automate more tasks. We therefore begin by thoroughly and systematically understanding how visual analysis has been used so far in various areas of software engineering.

Research Question A. *What areas of software engineering have benefited from visual analysis and how?*
We tackle this RQ by conducting a systematic survey to help structure, curate, and unify the dispersed literature in this research area, and to understand how visual analysis approaches have been used in software engineering, what areas have benefited from visual analysis and their rationale for adoption, and what are the challenges reported when they were used. This would help shed light on the potential of these techniques, and make them more visible and accessible. Chapter 2 is dedicated to this RQ.

1.1.1 Testability

Testability refers to the extent to which a software, or a portion of it, is amenable to having its functionality tested [102, 215]. A non-testable software would therefore have no mechanism by which any form of test specification can be defined and executed. In contrast, a fully-testable software has the capacity to have all of its functions tested in every aspect.

One of the least testable aspects in web UI are *canvas elements* [5]. These elements enable interactive and rich web graphics, visualizations, and charts. However, existing web testing technologies do not apply to these elements, thereby rendering them untestable. The reason behind this is that the state of canvas elements is not *observable* [5], by design, from a code-based perspective. That is, the canvas HTML element is just an empty placeholder that does not contain any information about the canvas, and therefore the canvas state is only *visually* observable by developers. The lack of observability has been also shown [102] to be one of the main factors affecting software testability in general.

Research Question B. *How can we make untestable web UI canvas elements testable through visual analysis?*
Chapter 3 is dedicated to investigating this RQ, where we propose a technique that visually analyzes the rendered web canvas elements and infers their states. The state describes the visual content of the canvas, their properties, and arrangement. This state is then synthesized and augmented into the DOM of the canvas element, thereby making it testable by existing web testing technologies (e.g., Selenium).

1.1.2 Accessibility

An accessible UI is one that allows users with disabilities, often visual disability [266], to easily use the various functionalities in the app. An inaccessible UI, on the other hand, can only be used by sighted users. Millions of people around the world are impacted by the lack of software accessibility [59]. Furthermore, many jurisdictions around the world are increasingly ratifying legal requirements for software to be accessible [204, 253].

Testing a web UI's accessibility requires ensuring that any information expressed through the visual structure and content of the UI has also been explicitly declared in the UI's markup. A number of tools exist for performing elementary *syntax* checks [256]. These checks provide quick and simple tests that, for instance, checks that no hyperlink elements are empty. They do not, due to their limited syntactic nature, test for the essential aspects of accessibility that require a *semantic* understanding the content and structure of the UI. There has been little to no research on automating the testing of these key qualitative and semantic aspects of accessibility, such as declaration of the high-level purpose of UI regions (e.g., navigation areas, menus), or specification of accessible form labeling, as will be discussed in Chapter 5. Therefore, the standard practice is based on the laborious time-consuming task of manually checking the UI for accessibility [9, 26].

Research Question C. *How can we visually test the semantic accessibility of web UIs?*

In Chapter 4, we address this RQ and propose a technique that performs a visual inference process to determine the semantic role of various regions in the UI. The markup of the UI is then tested to ensure that the visually inferred roles have been equivalently expressed in the markup.

Next, we explore another accessibility research problem related to web forms. It has been shown that missing *form labeling* is one of the three most common web accessibility issues [122]. Web form labeling accessibility requires explicitly declaring the labels of all fields, in order to allow non-sighted users to access that information. Accordingly, if a non-sighted user reaches a form that has no accessibility labeling, they currently have little to no available options in terms of automatically repairing the form labeling in order to be able to access that form. This, of course, can be a significant obstacle because web forms are ubiquitous.

Research Question D. *Can we automatically repair the accessibility of web UI forms through visual analysis?*
Chapter 5 explores this RQ. We propose a technique that abstracts web forms and builds a collection of visual cues for all form fields. An optimization problem is then formulated that infers the labeling associations for all fields. Finally, the inferred associations are augmented in the form's markup to ensure that the labeling is accessible.

1.1.3 Maintainability

Maintainability is defined as the degree to which software can be modified by its maintainers [3, 87]. It reflects the ease with which software can be modified, comprehended, or reused. Software has poor maintainability if it takes a lot of effort and time for developers to modify it in response to changes (e.g., changing requirements, environments). When structured using best practices [97, 116] (e.g., modularity, refactoring), software becomes more maintainable.

For web UI, creating *components* is a common approach that has been used in practice to improve maintainability [243]. A component is a piece of code that captures common or repetitive features in a UI, such that the component can be easily instantiated or replicated wherever needed [106]. However, creating components is often time consuming and tedious [210]; it requires several manual steps, including the examination of a UI mockup, checking potential elements that may or may not be suitable for conversion to components, constructing a template for components that unifies repeated segments, adding placeholders for variable content, and refactoring the code to replace instances with instantiated components.

Research Question E. *How can we generate reusable web UI components through visual analysis?*
Chapter 6 is dedicated to investigating this RQ. We propose a technique that visually analyzes common patterns in UI mockups. It then identifies how to group such repetitions into components. Finally, reusable UI components are generated and refactored into the page using common UI frameworks (e.g., React, Angular).

Chapter 2

Systematic Survey

2.1 Introduction

All areas of the software engineering (SE) lifecycle, such as requirements, design, development, and testing, often have the ultimate goal of contributing to a fundamental product of software engineering: the source code. Accordingly, a wide range of software engineering activities have typically revolved around the source code, whether to improve its quality, reliability, maintainability, or increase developers' productivity. A relatively more recent, and scarcely explored, alternative is the adoption of a *visual analysis* perspective. This approach aims to extract, analyze, or process visual aspects pertaining to the software, often using computer vision techniques. The objective is still focused on solving a software engineering problem, but is achieved via analyzing visual aspects of the software instead of relying exclusively on the source code. As an example, a typical visual analysis might involve comparing a couple of screenshot images in order to compare or analyze two graphical user interfaces (GUI) for testing purposes.

Visual analysis techniques have yielded promising results in developing robust and accurate solutions for various tasks. For instance, they have been successfully adopted to improve regression testing of GUIs [15, 60, 161], to identify cross-browser incompatibilities in web pages [220, 230, 231], to perform bug detection and automated program repair [168, 245], or to simplify software requirements modelling [155, 227].

In this chapter, we survey the literature on the use of visual analysis in performing software engineering tasks. Our work highlights visual analysis techniques and perspectives of addressing research topics in software engineering, what benefits they may provide compared to existing approaches, and what limitations they might bear. We believe this can be helpful in providing a distilled and concise overview of visual approaches in software engineering, building a concrete understanding of the advances made, and synthesizing insights regarding future directions for the research community. We conducted the survey by formulating a number of research questions to fulfill the goal of the study; we then proceeded by systematically collecting a pool of publications, and applied a number of inclusion and exclusion criteria. Subsequently, we analyzed and synthesized the collected papers by taking into account a number of dimensions, such as what area of software engineering (e.g., testing, maintenance) they brought benefit to, what specific task is being addressed (e.g., regression testing), what specific computer vision (CV) techniques have been used, and what is the rationale for their adoption.

2.2 Prior Work

To the best of our knowledge, there are no existing surveys or systematic literature reviews that share a similar goal to this chapter. This section discusses a broader range of research areas in order to put the survey in its proper context. We therefore begin by discussing secondary studies concerning computer vision-based techniques in various non-software engineering fields. While such studies are not directly related to our topic of discussion, they help place the survey within wider context of other similar surveys. Second, we explore other surveys that are interdisciplinary in nature, given the interdisciplinary nature of our survey. Finally, we discuss visual GUI testing techniques, an area with a relatively large number of papers employing visual analysis techniques.

Surveys on Computer Vision-based Engineering. In this subsection, we discuss some of the surveys or systematic literature reviews concerning the use of computer vision in various engineering fields. We note that these are non-software engineering fields (e.g., aerospace or automotive engineering) and are included here for the sake of completeness.

Kumar [144] catalogues the fabric defect detection methodologies reported in about 150 references into three main categories: statistical, spectral and model-based. They conclude that despite the significant progress in last decade, the problem of fabric defect detection still remains challenging and requires effort by combining existing approaches. Kanellakis and Nikolakopoulos [134] present a comprehensive literature review on vision based applications for unmanned aerial vehicles (UAVs) focusing mainly on current developments and trends. Computer vision techniques are used mainly for visual localization and mapping, obstacle detection and avoidance, aerial target tracking, and guidance. Among the limitations, it is mentioned that the algorithms are based on rigid assumptions such as low speed vehicles that do not account for fast scene alterations. Thus, the main challenge is to design solutions that can quickly react to ever changing sceneries, characterized by a high degree of dynamism and evolution. Liu and Dai [163] discuss solutions for UAVs from three main families, namely visual navigation, aerial surveillance and airborne visual simulation. Al-Kaff et al. [13] provide another survey of techniques for UAVs, particularly visual navigation algorithms, obstacle detection and avoidance and aerial decision-making. It is mentioned that artificial perception applications have represented important advances in the latest years in the expert system field related to unmanned aerial vehicles.

Gandhi and Triveli [100] discuss the recent research on pedestrian detection and collision prediction. Among the information gathered by the various sensors, the camera's image is one of the most used, along with visual analysis techniques for behaviour modelling in accident prediction, direction estimation, and collision prediction. Brunetti et al. [49] discuss vision-based pedestrian detection systems pertaining to three different application fields: video surveillance, human-machine interaction and analysis. Notably, they discuss both the differences between 2D and 3D vision systems, and indoor and outdoor systems. Janai et al. [125] provides a comprehensive survey on problems, datasets, and methods in computer vision for autonomous vehicles. First, they overview the datasets and benchmarks used in autonomous driving research. Then, the discuss the state of the art on several relevant topics, including recognition, reconstruction, motion estimation, tracking, scene understanding, and end-to-end learning.

Interdisciplinary Surveys in SE. Interdisciplinary surveys are often used to collect and analyze a body of knowledge across the boundaries between two or more fields. Here, we discuss some of the surveys or systematic literature reviews that have analyzed scientific and social fields from a software engineering perspective.

Zhang et al. [284] provide a comprehensive survey of techniques for testing machine learning systems. The survey covers 144 papers on different testing properties such correctness, robustness, and fairness, testing components (e.g., data, learning program, and frameworks), testing workflow (e.g., test generation and test evaluation), and application scenarios (e.g., autonomous driving, machine translation). The paper also analyses trends concerning datasets, research trends, and research focus, concluding with research challenges and promising research directions in machine learning testing. Beszédes [41] performed a systematic analysis of fault localization literature across different engineering fields, with the aim to find solutions in non-software areas that could be successfully adapted to software fault localization. Among their findings, they indicate that some classes of methods in computer networks literature are good candidates for adaptation, and could potentially be reused for software fault localization. Van der Linden and Hadar [258] performed a systematic literature review of physics of notation applications, a conceptual modelling language used for requirement specification. They analyzed what notations have been evaluated and designed using the physics of notation, for what reasons, to what degree applications consider requirements of their notation's users, and how verifiable these applications are.

Sabaren et al. [223] conduct a systematic literature review of cross-browser regression testing. In their survey, their goal was to collect the various techniques that have been proposed to perform cross-browser testing. The authors also describe several challenges in this specific context, such as the automatic identification of dynamic components in a user interface, which undermines the effectiveness of proposed testing techniques, causing many false positives in practice. We note that the survey of Sabaren et al. [223] has found 11 papers that happened to be in our final pool of 66 collected papers. This is a happenstance since our survey has a different objective for the following reasons. The work by Sabaren et al. [223] answers the following question: what approaches have been used to conduct cross-browser regression testing. In contrast, our work is not concerned at all with that problem. Our work answers the following question: in what ways have computer vision been used to advance software engineering. The reason we had some common papers is because regression testing happened to be an area where visual techniques were found to be particularly useful. However, in terms of the scope and objective, there is no overlap. In other words, Sabaren et al. [223] focus on a specific problem (i.e., cross-browser regression testing), regardless of what approaches were used (e.g., DOM analysis, state space navigation, visual analysis). That is, the survey in Sabaren et al. [223] is *problem-specific* but approach-agnostic. In contrast, our survey is *approach-specific* but problem-agnostic. Any overlap between the two surveys is because some papers happen to be in the pool of both surveys, but not because both surveys perform the same objective or have the same research questions. We focus on a specific approach (i.e., computer vision techniques), but consider its potential for any area of software engineering (e.g., testing, maintenance, development, design, requirements). In summary, none of the aforementioned surveys have a similar goal as that of this chapter.

Visualization Research. Visualization is the process of creating diagrams, charts, or any other kind of representation, from a given dataset. Visualization is part of any scientific process regardless of the field, and therefore has also been used in software engineering. There are a number of surveys on the use of visualization in various aspects of software engineering, such as surveys on visualization for software security [260, 285], surveys on visualization for static analysis [58], development coordination [246], maintenance and evolution [139, 196], to name a few. Visualization, however, is not the scope of this survey.

Visual GUI Testing. Issa et al [124] first introduced the notion of *visual testing* as a subset of traditional GUI testing. In their analysis, the authors conducted a study of bugs in four open source systems, and found that visual bugs represent between 16% and 33% of reported defects in those systems. In recent years, researchers and practitioners have started conducting empirical experiments aiming at understanding the comparative performance of a few visual testing approaches. For instance, Alégroth et al. [14, 16] present a case study of the benefits and challenges of using visual GUI testing by the team at one software company. In another study, Alégroth et al. [17] study the applicability and feasibility of Visual GUI testing in an industrial Continuous Integration environment, describing the main challenges faced by researchers to make it effective in practice. Garousi et al. [101] compare two popular visual testing tools (Sikuli and JAutomate) in one industrial project, and go through differences in test creation process, execution, and maintenance.

All such works analyze different technical and social aspects related to the use of Visual GUI testing in a specific context (i.e., the development and maintenance of test code). In contrast, our work is agnostic to any specific area or context. That is, it does not aim to focus on GUI testing. Rather, the goal is to broadly examine the use of visual techniques across any software engineering area (e.g., requirements, design, development, testing, maintenance) and for any task (e.g., refactoring, reverse engineering, regression testing).

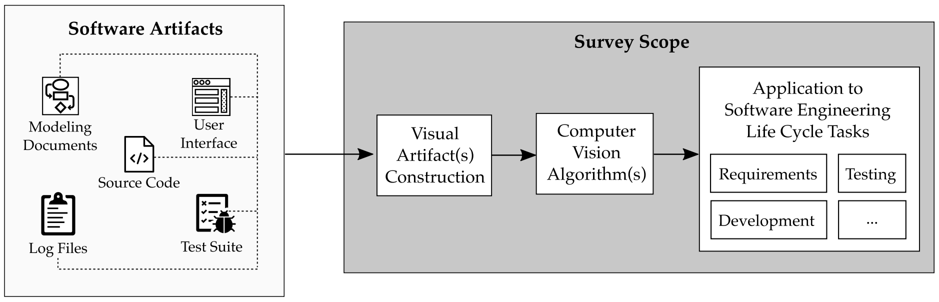


Figure 2.1: Overview of the scope of this survey.

2.3 Methodology

In order to conduct the survey in a thorough and structured manner, we follow the established guidelines by Kitchenham et al. [138]. We begin by introducing terminology and concepts needed to understand the remainder of this survey. Next, we define the scope of the work and flesh it out into specific research questions we aim to answer in this survey. We then describe the details of the paper collection process. Finally, we specify the inclusion and exclusion criteria applied to select the most relevant body of work from the existing literature.

2.3.1 Definitions

In order to categorize the use of visual approaches in software engineering, we use the terms *areas* and *tasks*. Software engineering areas are the various stages in the software lifecycle [239]. Examples of SE areas include software requirements, software design, and software testing. Within each area, different *tasks* can be defined. Each task is a specific activity that aims to achieve a well-defined objective related to that area. For instance, we refer to unit testing or regression testing as SE tasks within the software testing SE area, whereas code migration or code refactoring are tasks within software maintenance area. Accordingly, the rationale for using these two terms is to discuss our findings in more precise levels of granularity, in order to be able to analyze the findings across areas and for tasks within a specific area.

Next, we define the following terms in order to clarify which aspect of computer vision is being discussed:

Definition 1 (Visual Artifact) *A visual artifact is any datum that satisfies the following two conditions: (1) it constitutes a digital image or video, and (2) it is associated with one or more software engineering area(s).*

Definition 2 (Visual Approach) *A visual approach is an algorithm designed to solve a software engineering problem, which takes as input one or more visual artifacts, then typically incorporates a computer vision method or similar techniques as one or more of its steps, and finally yields an output that is used to achieve a software engineering task.*

The rationale for defining these two terms is to have precision and clarity when describing how visual analysis is used to solve a software engineering problem. We use the term *visual approach* to indicate that the approach used to solve an SE problem is based on analyzing visual data pertaining to the software. We use the term *visual artifact* to refer to software artifacts that are visual in nature, to differentiate them from other software artifacts that are non-visual (e.g., log files, requirements documents). The link between the two terms is that visual artifacts are the visual data consumed by a visual approach. Similarly, a visual approach is the algorithm that needs visual artifacts as input. To clarify all of the aforementioned terms, we give a simple example. Consider the case of cross-browser testing, where the goal is to check whether a given web app is being rendered identically in different browsers. Visual approaches for cross-browser testing often take a screenshot of the app in a set of different browsers, and then visually compare the screenshots. In this case, screenshots are the visual artifacts used or extracted from the software, and screenshot image comparison is the visual approach used to solve the SE task of cross-browser testing.

2.3.2 Scope

The scope of this work is to conduct a survey to help structure, curate, and unify the dispersed literature in this research area, and to analyze how computer vision techniques have been used in software engineering, and what challenges were reported when they were used. This would help shed light on the potential of these techniques, and make them more visible and accessible.

Figure 2.1 illustrates the scope of this work in relation to the software engineering life cycle and other software artifacts. The figure should be viewed as a multi-step process, beginning with visual artifacts construction, and ending with an application to a software engineering task, as defined in Section 2.3.1. As shown in the figure, the scope is to survey the following aspects: (1) how are visual artifacts (defined in Section 2.3.1) constructed or acquired from the software? This aspect is included in the scope because constructing or acquiring visual artifacts is the first step in a computer vision processing pipeline, and therefore should be explored in order to have a well-rounded survey. (2) what computer vision algorithms are used to analyze or process the constructed visual artifacts? This aspect is included in the scope because examining the visual processing or analysis conducted in order to address a given paper's research questions yields insight into how visual techniques can be potentially applied to various tasks of software engineering. (3) what are the software engineering areas and tasks where visual approaches have been used?

The figure also helps clarify what areas are outside the scope of this survey. For instance, the scope is not concerned with works where computer vision techniques were not utilized, or works that do not use any visual artifacts. Section 2.3.4 fleshes out the scope into a detailed set of inclusion and exclusion criteria.

2.3.3 Research Questions

As discussed in Section 2.3.2, the scope is to survey the use of computer vision in solving software engineering problems. In this section, we flesh out the scope into the following specific research questions:

RQ1: What are the main software engineering areas and tasks for which computer vision approaches have been used to date? We formulate this RQ in order to build a high level picture of the areas of software engineering where computer vision approaches were used. This aims at identifying potential trends of areas with high adoption of computer vision and conversely areas where little to no computer vision approaches were used. This can help the software engineering community in identifying potential gaps in the utilization of computer vision for software engineering.

RQ2: Why are computer vision approaches adopted? We formulate this RQ in order to identify common rationales for using computer vision to solve software engineering problems. This understanding of why computer vision approaches were used can subsequently help identify new software engineering areas or tasks where similar problems and rationales exist and therefore potentially benefiting from computer vision approaches.

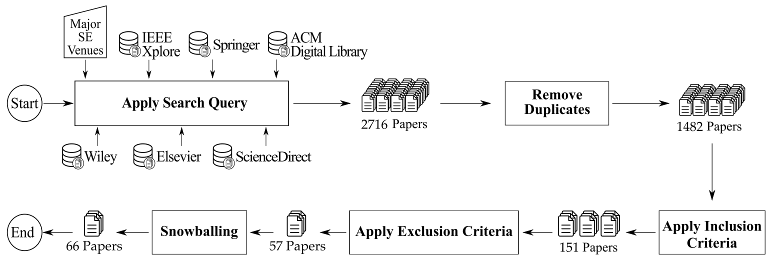


Figure 2.2: Overview of the paper collection process.

RQ3: How are computer vision approaches applied to software and its visual artifacts? This RQ is a natural progression of the previous RQs. The previous RQs identified the rationales of using computer vision and the SE areas where computer vision were used. This RQ examines the "how." That is, the mechanism(s) by which computer vision was applied to software. This can help guide the implementation of computer vision approaches to solve software engineering problems.

RQ4: How are software engineering tasks that leverage computer vision techniques evaluated? This RQ examines the methods used to evaluate the use of computer vision approaches in software engineering problems, as well as a summary of the reported limitations and challenges. This may help with selecting an evaluation strategy when exploring the use of a computer vision approach, and informing adopters of potential challenges.

2.3.4 Paper Collection

Figure 2.2 shows our paper search and selection process. In order to collect as much relevant literature as possible, we used two types of sources for paper collection: paper repository databases, and major software engineering venues.

Paper Repository Databases. To conduct our search, we used the databases of the following well-known publishers of scientific literature: IEEE Xplore, ACM

Digital Library, ScienceDirect, Springer, Wiley, and Elsevier. The search covers papers that have been published until June 2020. We used more than one database to ensure collecting as many papers as possible from all known publishers.

Software Engineering Venues. The preceding selection of paper repositories aims at casting a wide net in order to capture as many relevant literature as possible. However, since the databases contain an extremely large number of papers, it is possible that papers relevant to our survey are lost in the vast number of returned papers.

For this reason, and in order to make sure we collect highly relevant papers, we complemented the database search with a manual issue-by-issue search within the conference proceedings and journal articles from top-tier software engineering venues (listed in Table 2.1). The final pool of collected papers is the combined list of papers from both the database search and the SE venues search.

Interdisciplinary Venues. Given the interdisciplinary nature of this work, we also performed manual issue-by-issue search within the conference proceedings of relevant fields, namely, computer-human interaction, computer vision and machine learning. We selected the top three venues (based on the h5 index from Google Scholar) from each field. The searched venues are listed in the last section of Table 2.1 (under "Interdisciplinary Venues").

Search Query. For each of the aforementioned sources, we performed a search query using various combinations of terms to retrieve papers in different software engineering areas that are potentially using computer vision. The query was performed on all data fields of the paper, returning matches to either the *title*, *abstract*, *keywords*, or *text* of the paper. The query is composed of two parts: keywords related to the approach and keywords of the various software engineering areas. Keywords of the approach include strings such as “computer vision” or “visual”. They were included in the query in order to indicate our interest in works that use computer vision or visual approach. Keywords describing the areas include strings such as “development” or “testing” or any of the software engineering life cycle phases (e.g., requirements, maintenance). The final applied search query is as follows: *[computer vision OR image processing OR image analysis OR visual] AND [requirements OR design OR development OR testing OR maintenance OR comprehension]* The result of this query led to an initial pool of 2,716 papers, which was further filtered in the next stages.

Duplicates removal. During the paper collection step, we aimed to be thorough by including as many paper sources as possible in order to capture all potentially relevant works. However, this resulted in many duplicate papers since a given paper might be included in more than one database and venue. Therefore, we filtered the collected pool of papers by removing duplicate works based on their titles.

Inclusion and Exclusion Criteria

The search conducted on the databases and venues is, by design, very inclusive. This allows us to collect as many papers as possible in our pool. However, this generous inclusivity results in having papers that are not directly related to the scope of this survey. Accordingly, we define a set of specific inclusion and exclusion criteria and apply them to each paper in the pool, and remove papers not meeting the criteria. This ensures that each collected paper is inline with our scope and research questions.

Inclusion criteria. We define the following inclusion criteria: (1) The paper should be contributing to any stage of the software engineering process, whether in early requirements and modeling, through development and design, or finally testing and maintenance. We included this criteria in order to focus on software engineering papers. This is because we found a notable number of computer vision papers that were in fields other than software engineering (e.g., biology). (2) The paper should include a computer vision processing of the software or its artifacts. That is, the work achieves its objective (whether partially or fully) by extracting, analyzing, or processing visual artifacts pertaining or relevant to the software. This is an important and key criterion for paper selection because it ensures we meet the core scope of our survey. (3) The paper should be a full technical research paper that has a detailed description of the visual approach utilized. This criterion is imposed in order to have sufficient information to answer our research questions. Answering our research questions, such as RQ3 and RQ4, requires that we examine technical software engineering research papers, as opposed to, for instance, technical magazine articles, industrial white papers, or similar grey literature which do not have sufficient level of detail. Some demo/short papers can be allowed if they have dedicated and detailed sections discussing the detailed mechanism of the visual approach and its evaluation. This enables creating a pool of papers that have rich and detailed information and findings, in order to answer key research questions related to the details of the visual approach, the process of creating visual artifacts, and evaluation strategies. (4) The paper should contain a section dedicated to illustrating some form of quantitative or qualitative evaluation of the technique, or an illustration of its use case. This criterion was imposed in order to enable us to fully answer and explore our research questions regarding evaluation strategies.

These criteria were applied in a group review process by the authors. For each paper in the pool, each author initially checked the title and abstract, and briefly examined the proposed approach and results to ensure that it meets the inclusion criteria. If this check was not sufficient to conclusively decide whether the paper should be included in the pool, we proceeded with a secondary in-depth examination of the paper's objective, methodology, and evaluation to ensure that the inclusion criteria were met. Finally, a discussion among the authors was triggered, to decide on the inclusion of the paper in the final list of works.

Exclusion criteria. During our initial experimental test rounds of paper searching, we observed that a relatively large number of retrieved papers were on visualization research. This is understandable and expected because our queries include terms such as image, visual, and design.

Accordingly, we exclude papers published in the area of software visualization for the following reasons. First, the visualization class of algorithms does not constitute a *visual approach*, as defined in Section 2.3.1. Visualizations do not use any visual artifact of the software *as an input*. Rather, such work performs a final visual output or visual representation of a complete, non-visual, approach. Accordingly, this area of research is excluded since it would be outside the scope of this work. We recall that the scope is to survey visual approaches which, by definition, *consume* visual artifacts pertaining to the software during the course of running their algorithm or processing. Second, in addition to visualization being outside the scope of this survey, it is already a well-known and common aspect of software engineering, and plenty of surveys already exist on the use of visualization in various aspects of software engineering, as mentioned in Section 2.2.

Table 2.1: Conference proceedings and journals considered for paper collection (in addition to database search).

	Acronym	**Venue**
SE Conferences	ICSE	International Conference on Software Engineering
	FSE	International Symposium on Foundations of Software Engineering
	ESEC/FSE	Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
	ASE	International Conference on Automated Software Engineering
	ESEM	International Symposium on Empirical Software Engineering and Measurement
	ICST	International Conference on Software Testing, Verification and Validation
	ISSTA	International Symposium on Software Testing and Analysis
	MSR	International Conference on Mining Software Repositories
	RE	International Requirements Engineering Conference
	ICSME	International Conference on Software Maintenance and Evolution
	MODELS	International Conference on Model Driven Engineering Languages and Systems
	ISSRE	International Symposium on Software Reliability Engineering
	EASE	Evaluation and Assessment in Software Engineering
SE Journals	TSE	Transactions on Software Engineering
	EMSE	Empirical Software Engineering
	TOSEM	Transactions on Software Engineering and Methodology
	JSS	Journal of Systems and Software
	JSEP	Journal of Software: Evolution and Process
	STVR	Software Testing, Verification and Reliability
	ASE	Automated Software Engineering
	IEEE SOFTWARE	IEEE Software
	IET SOFTW.	IET Software
	IST	Information and Software Technology
	SQJ	Software Quality Journal
Interdisciplinary Venues	CHI	Conference on Human Factors in Computing Systems
	CSCW	Conference on Computer-Supported Cooperative Work
	UbiComp	Conference on Pervasive and Ubiquitous Computing
	UIST	Symposium on User Interface Software and Technology
	NeurIPS	Conference on Neural Information Processing Systems
	ICLR	International Conference on Learning Representations
	ICML	International Conference on Machine Learning
	CVPR	Conference on Computer Vision and Pattern Recognition
	ECCV	European Conference on Computer Vision
	ICCV	International Conference on Computer Vision

We also exclude commercial software services or open-source tools that have no corresponding publication, for the following reasons. First, including services or tools that are not peer-reviewed would negatively impact our ability to conduct the survey because tools and services that are not backed by a publication do not include a detailed explanation of their approach. This reduces our ability to answer key research questions for this survey, such as what specific computer vision techniques were used, what is the visual artifacts construction process, and how were the computer vision algorithms applied to the visual artifacts. Services and tools without a corresponding publication also typically do not have a thorough systematic evaluation, and therefore we are unable to answer research questions related to how computer vision techniques were evaluated, what are the main findings, and what were the challenges.

Snowballing. At the end of searching database repositories and conference proceedings and journals, and applying inclusion and exclusion criteria, we obtained a total of 57 unique papers. Next, to mitigate the risk of omitting relevant literature from this survey, we also performed backward snowballing [269] by inspecting the references cited by the collected papers so far. Nine additional papers were retrieved during this phase, which led to a final pool of 66 unique papers. Table 2.3 shows the final pool of papers that will be discussed and analyzed in the remainder of this work.

Extracted Information. For each retrieved paper, we collect a set of data necessary to answer the research questions. Table 2.2 shows the list of data collected from each paper and their mapping to each research question. As shown in the table, the title, author(s), and document ID were used for documentation purposes to

Table 2.2: Collected data items (Synthesis Matrix)

Field	**Use**
Title	Documentation
Author(s)	Documentation
DOI identification number	Documentation
Abstract	Paper Selection
Text	Paper Selection
Venue	RQ1
Year	RQ1
Target platform	RQ1
Software engineering area	RQ1
Software engineering task	RQ1
Reasons for adopting CV approach	RQ2
Visual artifact(s) used	RQ3
CV algorithm(s) used	RQ3
Evaluation process & challenges	RQ4
Main results	RQ4
Limitations of CV methods used	RQ4

keep track of the various papers. The abstract and text were used for the paper selection process and applying the inclusion and exclusion criteria. The venue, year, software engineering area and task of each paper was also collected in order to discuss and answer RQ1. We also extract the target platform for each paper, which is the type of computing device (e.g., desktop, mobile) that the analyzed software runs on. A list of reasons for adopting computer vision was also extracted from each paper in order to answer RQ2. The visual artifact(s) and the visual approach utilized were also identified in each paper in order to discuss RQ3. Finally, we log the evaluation process and the results and findings from each collected paper in or-

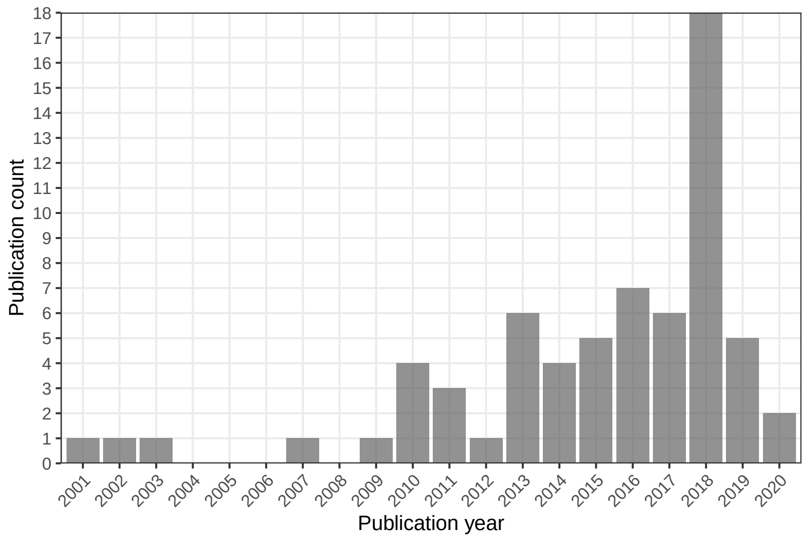


Figure 2.3: Distribution of publications across years.

der to answer RQ4. All these data are collected, analyzed, and used to synthesize the findings for the rest of this survey. In order to facilitate the use of this data by the general research community, the data have been made publicly available at http://tiny.cc/tse-2020.

2.4 Findings

2.4.1 Trends and Landscape

At the end of the paper collection process, we obtained a pool of papers spanning the years 2001-2020 (June 2020). Table 2.3 shows the final list of 66 papers. Figure 2.3 shows the distribution of the retrieved pool of papers across different years of publication. Overall, we observed a generally increasing long-term trend in the use of computer vision approaches in software engineering. This research area is also relatively new, with more than half of the papers in our pool published in the

Table 2.3: Collected pool of papers (in chronological order).

Reference	Title	Venue	Year
Landay and Myers [145]	Sketching Interfaces: Toward More Human Interface Design	IEEE Computer	2001
Caetano et al. [54]	JavaSketchIt: Issues in Sketching The Look of User Interfaces	AAAI	2002
Fails and Olsen [92]	A Design Tool for Camera-based Interaction	CHI	2003
Coyette et al. [73]	Multi-fidelity Prototyping of User Interfaces	INTERACT	2007
Zheng et al. [287]	Correlating Low-level Image Statistics with Users - Rapid Aesthetic and Affective Judgments of Web Pages	CHI	2009
Chang et al. [60]	GUI Testing Using Computer Vision	CHI	2010
Choudhary et al. [66]	WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications	ICSM	2010
Li et al. [155]	FrameWire: A Tool for Automatically Extracting Interaction Logic from Paper Prototyping Tests	CHI	2010
Dixon and Fogarty [82]	Prefab: Implementing Advanced Behaviors using Pixel-based Reverse Engineering of Interface Structure	CHI	2010
Delamaro et al. [80]	Using Concepts of Content-based Image Retrieval to Implement Graphical Testing Oracles	STVR	2011
Dixon et al. [83]	Content and Hierarchy in Pixel-based Methods for Reverse Engineering Interface Structure	CHI	2011
Seifert et al. [229]	Mobidev: A Tool for Creating Apps on Mobile Phones	MobileHCI	2011
Choudhary et al. [64]	Crosscheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications	ICST	2012
Givens et al. [103]	Exploring The Internal State of User Interfaces by Combining Computer Vision Techniques with Grammatical Inference	ICSE	2013
Liang et al. [156]	SeeSS: Seeing What I Broke – Visualizing Change Impact of Cascading Style Sheets (CSS)	UIST	2013
Scharf and Amma [227]	Dynamic Injection of Sketching Features Into GEF-based Diagram Editors	ICSE	2013
Alégroth et al. [15]	JAutomate: A Tool for System- and Acceptance-test Automation	ICST	2013
Semenenko et al. [231]	Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features	ICSM	2013
Roy Choudhary et al. [220]	X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications	ICSE	2013
Lin et al. [161]	On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices	TSE	2014
Mahajan and Halfond [168]	Finding HTML Presentation Failures Using Image Comparison Techniques	ASE	2014
Amalfitano et al. [19]	Towards Automatic Model-in-the-loop Testing of Electronic Vehicle Information Centers	WISE	2014
Selay et al. [230]	Adaptive Random Testing for Image Comparison in Regression Web Testing	DICTA	2014
Bao et al. [34]	scvRipper: Video Scraping Tool for Modeling Developers' Behavior Using Interaction Data	ICSE	2015
Nguyen and Csallner [193]	Reverse Engineering Mobile Application User Interfaces with REMAUI	ASE	2015
Burg et al. [51]	Explaining Visual Changes in Web Interfaces	UIST	2015
Mahajan and Halfond [169]	Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques	ICST	2015
Hori et al. [117]	An Oracle based on Image Comparison for Regression Testing of Web Applications	SEKE	2015
Reinecke et al. [211]	Enabling Designers to Foresee Which Colors Users Cannot See	CHI	2016
Deka et al. [78]	ERICA: Interaction Mining Mobile Apps	UIST	2016
Ponzanelli et al. [203]	Too Long; Didn't Watch! Extracting Relevant Fragments from Software Development Video Tutorials	ICSE	2016
Mahajan et al. [171]	Using Visual Symptoms for Debugging Presentation Failures in Web Applications	ICST	2016
Feng et al. [94]	Multi-objective Test Report Prioritization Using Image Understanding	ASE	2016
Patrick et al. [202]	Automatic Test Image Generation Using Procedural Noise	ASE	2016
He et al. [115]	X-Check: A Novel Cross-browser Testing Service Based on Record/Replay	ICWS	2016
Deka et al. [79]	Rico: A Mobile App Dataset for Building Data-Driven Design Applications	UIST	2017
Wan et al. [262]	Detecting Display Energy Hotspots in Android Apps	STVR	2017
Bao et al. [35]	Extracting and Analyzing Time-series HCI Data from Screen-captured Task Videos	EMSE	2017
Zhang et al. [283]	Sketch-guided GUI Test Generation for Mobile Applications	ASE	2017
Chen et al. [62]	UI X-Ray: Interactive Mobile UI Testing Based on Computer Vision	IUI	2017
Wu et al. [271]	Automatic Alt-text: Computer-generated Image Descriptions for Blind Users on a Social Network Service	CSCW	2017
Reiss and Miao [212]	Seeking the User Interface	ASE J.	2018
Kıraç et al. [140]	VISOR: A Fast Image Processing Pipeline with Scaling and Translation Invariance for Test Oracle Automation of Visual Output Systems	JSS	2018
Leotta et al. [154]	Pesto: Automated Migration of DOM-based Web Tests Towards the Visual Approach	STVR	2018
Bajammal and Mesbah [30]	Web Canvas Testing through Visual Inference	ICST	2018
Xu and Miller [275]	Cross-Browser Differences Detection Based on an Empirical Metric for Web Page Visual Similarity	TOIT	2018
Kuchta et al. [143]	On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files	EMSE	2018
Bao et al. [36]	VT-Revolution: Interactive Programming Video Tutorial Authoring and Watching System	TSE	2018
Moran et al. [185]	Automated Reporting of GUI Design Violations for Mobile Apps	ICSE	2018
Chen et al. [61]	From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation	ICSE	2018
Sun et al. [248]	Neural Program Synthesis from Diverse Demonstration Videos	ICML	2018
Lim et al. [158]	Ply: A Visual Web Inspector for Learning from Professional Webpages	UIST	2018
Moran et al. [187]	Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps	TSE	2018
Stocco et al. [245]	Visual Web Test Repair	FSE	2018
Tanno and Adachi [252]	Support for Finding Presentation Failures by Using Computer Vision Techniques	ICST	2018
Bajammal et al. [32]	Generating Reusable Web Components from Mockups	ASE	2018
Moran et al. [186]	Detecting and Summarizing GUI Changes in Evolving Mobile Apps	ASE	2018
Natarajan and Csallner [191]	P2A: A Tool for Converting Pixels to Animated Mobile Application User Interfaces	MOBILESoft	2018
Osman et al. [198]	An Automated Approach for Classifying Reverse-engineered and Forward-engineered UML Class Diagrams	SEAA	2018
Xiao et al. [274]	Automatic Identification of Sensitive UI Widgets based on Icon Classification for Android Apps	ICSE	2019
Huang et al. [119]	Swire: Sketch-based User Interface Retrieval	CHI	2019
Zhao et al. [286]	ActionNet: Vision-Based Workflow Action Recognition from Programming Screencasts	ICSE	2019
Yu et al. [281]	LIRAT: Layout and Image Recognition Driving Automated Mobile Testing of Cross-Platform	ASE	2019
Swearngin and Li [249]	Modeling Mobile Interface Tappability using Crowdsourcing and Deep Learning	CHI	2019
Yuan and Li [282]	Modeling Human Visual Search Performance on Realistic Webpages using Analytical and Deep Learning Methods	CHI	2020
Wu et al. [273]	Predicting and Diagnosing User Engagement with Mobile UI Animation via a Data-Driven Approach	CHI	2020

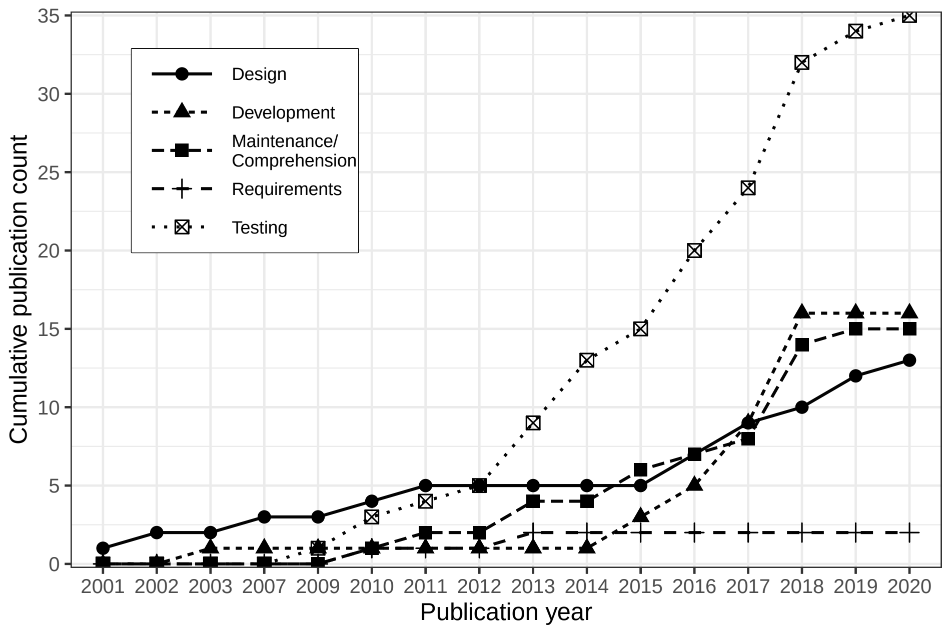


Figure 2.4: Cumulative distribution of publications across years per SE area. Testing is the most common area, followed by development and maintenance.

past five years. Furthermore, Figure 2.4 depicts the cumulative number of publications per software engineering area across different years. The results indicate that software testing is the area exhibiting the most rapid increase in terms of the number of publications wherein a computer vision technique is utilized.

Figure 2.5 shows the distribution of the published papers across venues. The main venues in which computer vision approaches for software engineering were published are the Conference on Human Factors in Computing Systems (CHI) with 11 papers, the International Conference on Software Engineering (ICSE) with nine papers, and the International Conference on Automated Software Engineering (ASE) with eight papers. The presence of traditional SE venues as well as venues

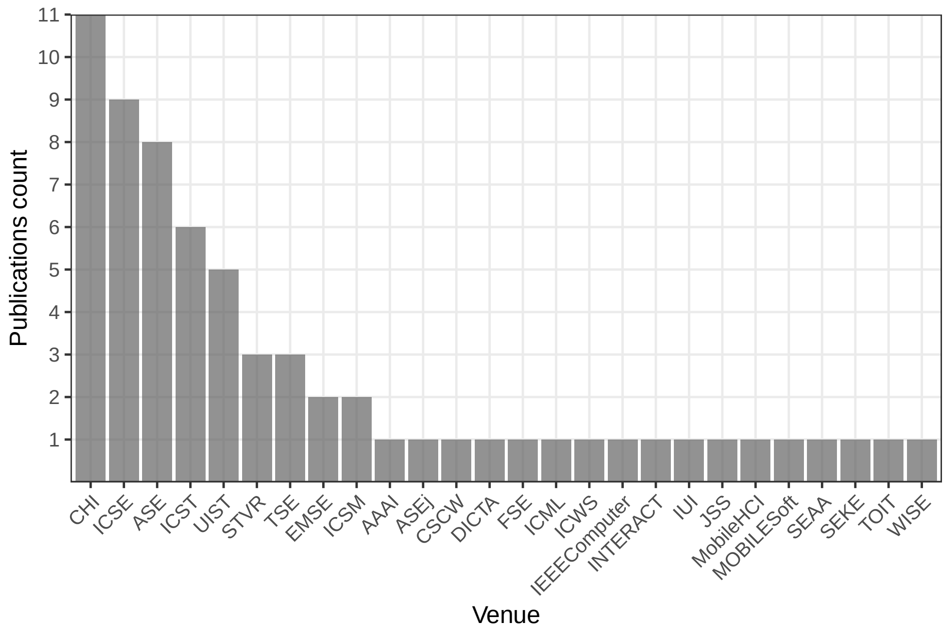


Figure 2.5: Distribution of the publications across venues.

from other fields (e.g. CHI) in Figure 2.5 provides some indication that research on the use of computer vision for software engineering tasks is an interdisciplinary field.

2.4.2 Areas, Tasks, and Platforms (RQ1)

To study the exiting applications of visual techniques for SE, we analyzed the selected papers to find out which *SE areas* have been explored, for which *tasks*, and on which *platforms* they were used. As defined in section 2.3.1, SE areas are high-level stages of the software engineering life cycle, such as requirements, testing, or development. SE tasks are more fine-grained activities, such as unit testing or regression testing. The platforms are the types of computing devices (e.g., desktop, mobile) that the analyzed software runs on. We further looked into the papers' discussion sections to gain insights from the authors about other areas in which the proposed technique could potentially be applied.

Software Engineering Areas and Tasks

Figure 2.6 presents the papers distribution across different SE areas and tasks. Note that the number of papers indicated in the figure is more than the total number of papers in the pool. This is due to the fact that for some papers, the presented approach can be utilized for more than one task. We now discuss more in detail the trends of Figure 2.6.

Testing. Software testing is the most common research area for which approaches using computer vision are proposed, accounting for approximately half of all collected papers. A closer look at the publications in this area reveals interesting trends. **Regression Testing.** Most of the studies use visual methods to facilitate acceptance and regression testing e.g., by comparing visual artifacts (e.g., the GUIs) with each other or with respect to a given oracle. Without adopting computer vision, developers would most likely need to perform some kind of manual evaluation, e.g., through eyeball analysis to spot deviations from the expected visual presentation—a daunting and error-prone task. Apart from GUI comparisons, computer vision (CV) techniques have been also utilized for other software testing tasks. For example, Kuchta et al. [143] introduce a technique for regression testing of PDF reader software and localizing faulty parts of PDF files. The adopted technique exploits *differential testing*, where the (visual) output of multiple implementations of a program—the PDF viewer—is compared to the same input to spot deviations. In another work, Leotta et al. [154] propose an automated code migration tool for automatically converting end-to-end Selenium web tests to vi-

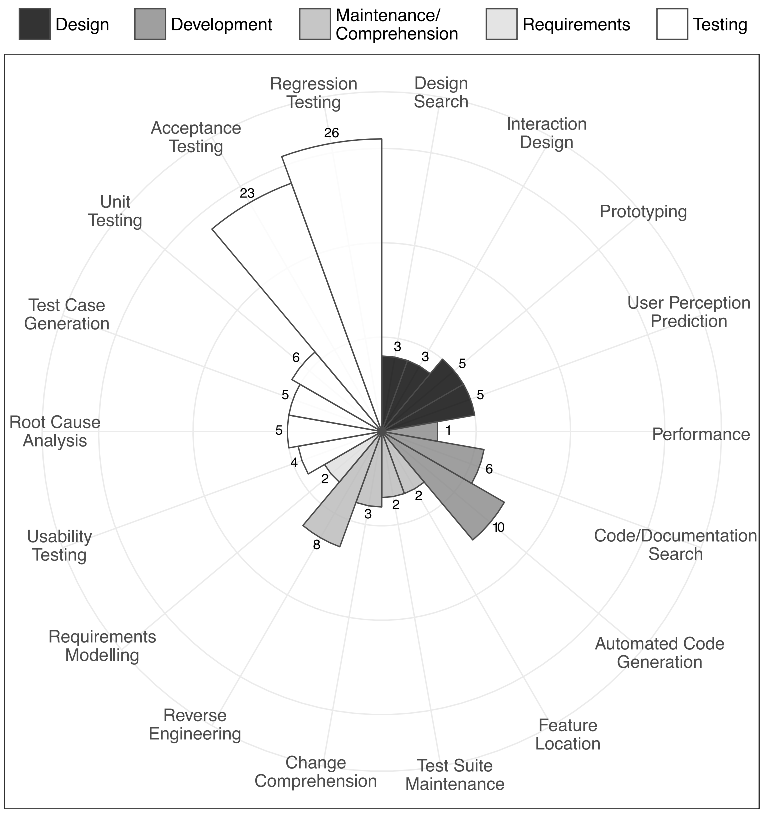


Figure 2.6: Papers distribution across different Software Engineering Areas and Tasks

sual web tests based on Sikuli's [60] image recognition capabilities. Kıraç et al. [140] provide an image comparison technique for black-box, regression testing of visual-output software used in consumer electronics. The approach removes noise to eliminate image differences caused by scaling and translation, and is evaluated on the output of digital TVs. Bajammal and Mesbah [30] propose a technique to test state-free canvas elements on web pages by reverse-engineering a visual model. This allows unit testing of the specific visual elements contained in the canvas.

Cross-browser Testing. A significant class of software testing techniques leveraging visual methods aims at automatically identifying *cross-browser incompatibilities* (XBIs) for web applications [64, 66, 115, 220, 230, 231, 275]. XBIs are frequently-occurring issues in web pages' appearance and/or behavior when the same page is viewed on different web browsers [220]. Identifying such differences requires laborious human judgement, which can be effectively reduced using an automated visual-based technique. A recent literature review by Sabaren et al. [223] surveys the techniques proposed to tackle XBIs.

A similar problem involves using visual methods for *root cause analysis* of presentational issues occurring on web pages [168, 169, 171]. In addition to the web domain, root cause analysis is also used to identify rendering issues in PDF files [143], as well as, the causes of excessive energy consumption of UIs in mobile applications [262].

Several approaches attempt to automate test execution using visual techniques. An example is *record-and-replay testing* where the screenshots of the GUI and the human tester's actions and inputs performed on the GUI are recorded [15, 60, 115, 161] . Popular visual record-and-replay tools are Sikuli [60] and JAutomate [15], which allow fast and easy replay of the same sequence of actions: the tool conducts a visual search on the current visible contents of the screen to detect the widgets' locations, triggers the recorded actions and inputs, and finally performs a visual assertion, comparing the observed visual outcome with the expected oracle.

Besides record-and-replay tools, test automation with visual analysis has been successfully applied to *automotive software engineering*. Amalfitano et al. [19] propose a tool to automate the testing of the emulated vehicle information systems' panels. The testers can locate visual elements on the panel and specify their properties; the tool allows to check the panel's output with respect to these properties at pre-defined timestamps.

Zheng et al. [287], Yuan and Li [282], Deka et al. [78], and Deka et al. [79] aim to automate the testing of *aesthetics or usability* of web pages. This line of work involves building computer vision models that can predict whether web pages meet certain aesthetic requirements pertaining to the usability of a page (e.g. visual balance of white space and elements, consistent and simple representation of elements on a web page).

Patrick et al. [202] propose an approach based on systematic image manipulation to *automatically generate test input images* for regression and acceptance testing of an epidemiological simulation software. The software's output on these input images is monitored and unexpected deviations reveal bugs or regressions in the code.

To *prioritize test reports*, Feng et al. [94] use the screenshots provided by users to augment the existing textual test prioritization techniques for mobile applications. Finally, to *automatically generate test cases*, Zhang et al. [283] use a visual-aided approach that identifies strokes that testers draw on screenshots taken from the apps. These sketches are used to define test specifications (e.g., coordinates where a visual object should be positioned to on the screen), which are subsequently used to generate test cases automatically.

Design. Our survey included a number of papers aiming at facilitating the design stage of software systems by means of computer vision. Li et al. [155] propose a tool to help with *prototyping* software designs. Using computer vision techniques, a video recording of hand-drawn GUI design sketches are converted into a digital form. This serves as an interactive, clickable, documentation of the prototype which can be easily shared with stakeholders. The works of Landay and Myers [145], Caetano et al. [54], Coyette et al. [73], and Scharf and Amma [227] also share the same goal of facilitating user interface prototyping by using computer vision to convert hand-drawn GUI sketches to working GUI prototypes.

Deka et al. [78] use a visual technique to learn features from mobile applications' UIs to create a database of UI design samples, forming a benchmark for *design searching*. In a successive work, Deka et al. [79] record the crowed-sourced *interactions* with the application's GUIs. This allows mining these interactions to incorporate them in new designs, and *predicting users' perception* by their interaction with new GUIs. The works of Reinecke et al. [211], Swearngin and Li [249], Wu et al. [273] also propose visual techniques to help UI designers in predicting the user perception of their designs. Reinecke et al. [211] visually examines the color spectrums and arrangements in a web page, and informs UI designers if certain demographics (e.g. color-blind users) would not be able to see certain parts of their designs. Swearngin and Li [249] build a computer vision model that mimics users perception of "tappability" of various elements in a mobile app. Accordingly, if an app designer has an element that is tappable, but would not be perceived as tappable for the average user, the tool would flag such elements. Wu et al. [273] focuses on flagging animations that would be perceived (by the average user) as too fast, chaotic, or lacking transitions. The UI designer is then notified of these issues in order to mitigate them.

Requirements Engineering. Requirements engineering was the least explored area among all collected papers, with only two publications. This outcome is understandable since it might be due to the increasing adoption of agile development compared to the more traditional waterfall model. Visual techniques have been used to generate a digital form of requirement or design models (e.g., UML) by visually processing hand-made sketches [227], or by augmenting existing requirement artifacts to make them user-tractable [155].

Comprehension and Maintenance. Visual techniques have been used in software *reverse engineering*. The REMAUI tool [193] uses computer vision techniques to reverse engineer the UI elements and their hierarchy in a mobile application from a screenshot (or a mockup), which also allows to automatically generate the UI code. Givens et al. [103] performs a similar reverse engineering of the internal state of desktop applications based on visual decomposition of screenshots. Dixon et al. [83] takes this a step further by reverse engineering the hierarchy of interface components. Bajammal and Mesbah [30] reverse engineers the state of web canvas elements from a visual screenshot of the canvas itself, which also enables testing of canvas elements. Deka et. al. [78], [79] captures traces of user's interaction with mobile apps, allowing the mining of user interactions from a large collection of apps.

In another work, Burg et al. [51] use visual techniques for localizing the JavaScript code responsible for the implementation of a single widget that determines an interactive behaviour on a web application (i.e., *feature location*). Lim et al. [158] presents a similar tool but focuses on localizing the CSS implementation responsible for certain visual appearances.

Stocco et al. [245] present a visual approach for *automated test repair*; they propose a technique to repair broken web test cases by visually analyzing test executions. Finally, Leotta et al's approach [154] for migrating Selenium-based web test cases to Sikuli can help in *maintaining* web tests, i.e., when it is required to convert DOM-based locators (e.g., XPath expressions) to modern visual locators.

Development. Visual techniques have been used for *automated code generation*, simplifying the development stage of software engineering. This includes generating UI code from mockups [32, 61, 193], existing mobile apps UI code [78, 79], hand-made sketches [212], or from a video recording depicting the desired behavior [248]. Wu et al. [271] propose a tool that automatically annotates HTML images with suitable alternative texts.

Fails and Olsen [92] present a tool that helps developers in the creation of software that processes live camera feeds, without requiring developers to have computer vision skills. Bao et al. [34] proposes a similar tool that facilitates scraping of developers' screencast videos, which simplifies searching for code and documentation from video tutorials.

Reiss and Miao [212] propose a technique to *search code* from existing repositories based on a given sketch, to make a compilable code from the results. Ponzanelli et al. [203] allow searching relevant code fragments from video tutorials using visual techniques. Bajammal et al. [32] generate UI component code (e.g., React, AngularJS) from a visual analysis of a web app's mockup design. Finally, Wan et al. [262] allow to spot energy pitfalls in the UIs of mobile apps, allowing a more performance-aware UI development.

Platforms

Table 2.4 illustrates the results of our analysis with respect to the platforms in which visual approaches were utilized. More than half of the collected papers target web and mobile platforms.

Web and mobile applications are ubiquitous nowadays, and their sole communication interface with users is through their GUIs. Desktop applications, on the contrary, can often have different interfaces, e.g., a command-line interface, or a network interface where the use of an external client software is required. Hence, it is not surprising for visual approaches to be more utilized in web and mobile domains. However, there are also other interesting platforms [19, 140], (e.g. automotive dashboards or digital TVs) where visual techniques have been successfully applied. This indicates the potential of visual techniques in any platform where software deals with a GUI, or any artifact that is visual in nature.

Summary. This section focused on exploring the areas, tasks, and platforms where computer vision techniques have been proposed to address software engineering problems. We found that software testing is the most common SE area where computer vision techniques have been used. Within the area of software testing, cross-browser compatibility is the most frequent task that uses computer vision, and it also represents some the earliest works in the use of visual analysis for software engineering. Non-functional properties received little to no exploration, which is a research area that may potentially benefit from visual techniques due to the high level and semantic nature of non-functional properties. We also found that more than half of the collected papers target web or mobile platforms, as opposed to desktop.

2.4.3 Rationale (RQ2)

The goal of this RQ is to understand the *motivations* behind the use of visual approaches in the collected publications. For each paper in our pool, we analyzed the paper's full text and noted down the rationales mentioned by the authors for using computer vision to solve the software engineering problem being tackled. This resulted in three main categories, namely, *context-driven, ease of use,* and *robust-*

Table 2.4: Papers distribution across different platforms

Platform	**Papers**
Web	[30, 32, 51, 64, 66, 115, 117, 154, 155, 156, 158, 168, 169, 171, 220, 230, 231, 245, 271, 275, 282, 287]
Desktop	[15, 34, 35, 36, 54, 60, 73, 80, 82, 83, 92, 94, 103, 143, 145, 198, 202, 203, 211, 212, 227, 248, 286]
Mobile	[61, 62, 78, 79, 119, 161, 185, 186, 187, 191, 193, 193, 229, 249, 252, 262, 273, 274, 281, 283]
Other	[19, 140]

ness (as will be described below). For each paper that did not explicitly mention their rationale, we analyzed the text and classified the paper to the closest rationale category. We now describe the three identified categories of rationale.

Context-driven. Computer vision has been utilized because the context is intrinsically *visual* in nature, which is the case in all the papers that focused on GUIs. Thus, it was natural for the authors to deal with a visual artifact through a computer vision technique. More than half of the selected papers motivate the use of visual approaches as such [15, 19, 30, 35, 51, 60, 62, 78, 79, 80, 94, 115, 140, 143, 154, 155, 169, 171, 193, 202, 203, 212, 227, 230, 262, 275].

For instance, Chang et al. [60] describe two properties of visual approaches that make them particularly appealing for analyzing GUI-based software: *intuitiveness* and *universality*. They describe how for certain tasks, using visual artifacts—such as a GUI screenshot— is a more spontaneous way of interaction with the software. Due to their graphical nature, elements on the GUI can be most directly represented by screenshots. Non-visual alternatives, such as scripting, would instead require users to manipulate GUI elements through keywords which is arguably less intuitive.

Furthermore, screenshots are easily accessible for all GUI-based applications. Indeed, it is virtually always possible to take a screenshot of a GUI element, across all applications and platforms. This can make it attractive to propose techniques based on analyzing visual artifacts.

Ease of Use. As a second main motivation, researchers have utilized visual approaches because they deemed them *easier* to use by end users [19, 62, 64, 66, 80, 115, 143, 161, 168, 169, 171, 220, 230, 231, 275, 283]. For instance, Zhang et al. [283] propose a tool that allows developers to draw (e.g. via tablets, digital pens) simple sketches on app screenshots. The tool then uses CV algorithms to analyze the shapes and structure of these hand drawn sketches to decode the meaning of each sketch. The tool then uses the sketch as a visual test spec to automatically generate a number of GUI test suites for mobile applications. The authors argue, and demonstrate, that providing developers with the option of using simple hand sketches to automatically generate test cases is a more natural and easier to use approach to create test cases compared to manually writing the test cases.

This viewpoint is also evident in papers targeting the detection of cross-browser incompatibilities (XBIs) [64, 66, 115, 220, 230, 231, 275]. This problem requires developers to detect visual differences between web pages within the same web application when rendered on different browsers. This task is challenging for a number of reasons. First, manually performing this task, for instance through eyeballing, is neither efficient nor easy. Hence, an automatic technique would require simulating the reasoning that humans do while *seeing and comparing* two web

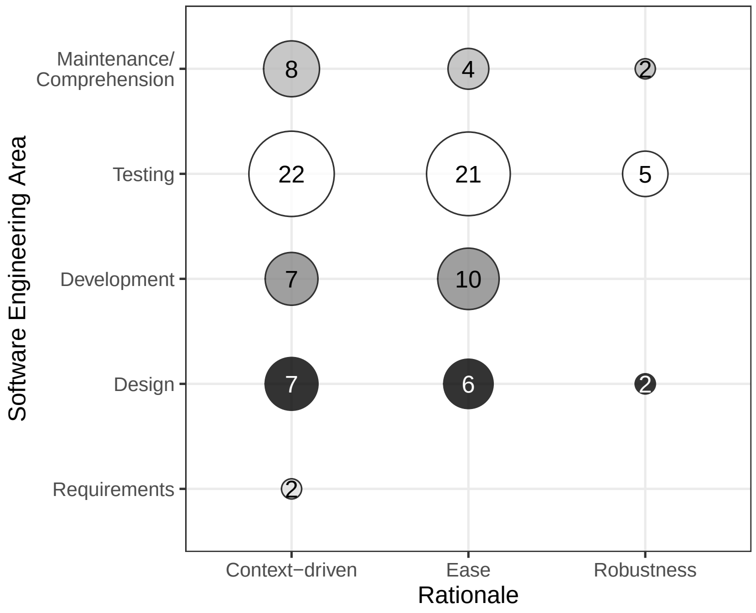


Figure 2.7: Distribution of rationales per SE area.

pages. This can be easily simulated through a computer vision technique called *image differencing*, for which a plethora of different techniques have been proposed [64, 220, 231].

However, originally, the same problem was tackled from a non-visual perspective. First works on XBIs used well-known DOM differencing techniques as a proxy for finding visual defects. The main limitation was the fact that DOM-level differences do not always correspond to a different visual layout. Hence, this caused such techniques to have many false positives and a low accuracy. On the other hand, visual approaches have been proposed both as an alternative, as well as, a complementary technique to overcome the limitations of the DOM-based approaches.

Robustness. The concept of robustness of a visual approach concerns its capability of maintaining its effectiveness despite minor visual changes happening in the software being analyzed.

A few papers [15, 60, 252] mention robustness as rationale for choosing computer vision. They explain that visual approaches are used because they are considered more change-tolerant than alternative code-based techniques. In other words, according to the authors, using a code-based approach for the same problem is likely to produce a fragile tool that would require a high maintenance cost.

For instance, Chang et al. [60] describe how spatial re-arrangements of GUI components on the page can lead to fragile test scripts, a well-known issue in web testing [151]. According to Chang et al. [60], visual approaches are more robust to minor layout changes and elements repositioning. This viewpoint is discussed and confirmed in the paper by Alégroth et al. [15], in which image recognition of GUI elements allows the development of more robust system-level automated test suites. A similar finding is in the paper by Stocco et al. [245], in which an image processing pipeline was used to automatically trace web elements across different versions of the same web applications. Their tool VISTA exhibited a high test repair rate during software evolution, outperforming a DOM-based test repair solution. This essentially means that in the web domain, web app GUIs exhibit less frequent changes as compared to the DOM, as acknowledged by other researchers [112, 113, 148, 150].

The bubble chart of Figure 2.7 shows the distribution of the rationales in relation to the SE areas presented in Section 2.4.2. We notice that the context-driven category dominates across all SE areas. In the areas of requirements, design, and maintenance, visual approaches were prevalently used because, at this stage of the software development lifecycle, designers or requirements engineers mostly deal with visual abstractions of (portions) the software such as GUI mockups, or UML models. Computer vision allows the transformation of these visual artifacts to support successive SE tasks. For example, in the work by Zhang et al. [283], annotated sketches are used to specify test requirements and test case creation. In this case, the targeted SE area is both requirements engineering and testing.

Summary. In this section, the goal was to understand the rationales and motivations for using computer vision to address software engineering problems. An understanding of the rationales can help researchers decide if their research area or topic has similar problems or challenges, and then potentially explore using computer vision for their problem. We identified three types of rationales by collecting and categorizing the reasons stated by the authors of each paper. There are context-driven, ease of use, and robustness. Papers were classified as context-driven when the context of the software engineering problem itself has dictated the use of visual approaches. The ease of use classification was used in cases where the motivation is not necessarily driven by the context, but driven by the motivation of making an existing software engineering process easier to use (e.g. has less manual work, easier to comprehend). Finally, papers were classified as motivated by robustness whenever the motivation is making a software engineering task more accurate or less fragile. Out of all three categories of rationales, the context-driven category was the most common.

2.4.4 Computer Vision Techniques (RQ3)

In order to investigate how computer vision is applied, we studied: (1) what visual artifacts are used, generated, or extracted from the software, and (2) what computer

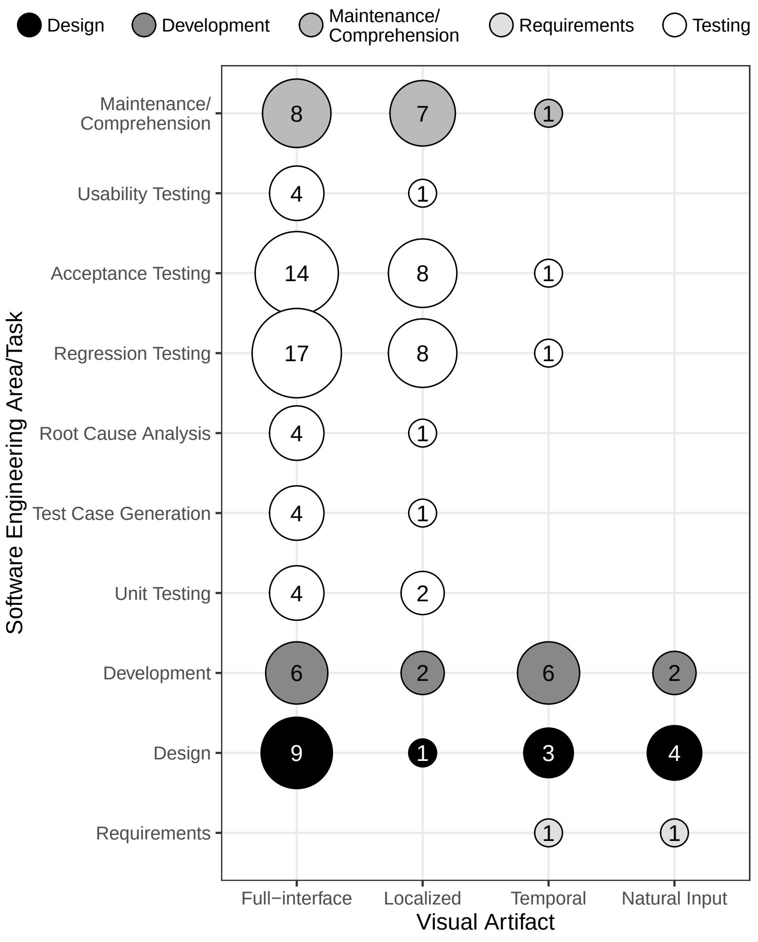


Figure 2.8: Distribution of visual artifacts per SE area & task.

vision techniques are used to process or analyze the visual artifacts. We recall that visual artifacts are visual data (e.g., images) used by one or more computer vision techniques, with the final objectives of addressing a software engineering problem.

Artifact Categories. Through our survey of the field, we classified the visual artifacts used in the literature into four categories: (1) full-interface artifacts, (2) localized artifacts, (3) temporal artifacts, and (4) natural input artifacts. Figure 2.8 shows the distribution of such visual artifacts with respect to the SE area/tasks.

The first category, *full-interface visual artifacts*, typically represents screenshots of the entire user interface of the system (whether a web browser or desktop application, as well as other forms such as visual content of TVs and car displays) [62, 64, 66, 78, 79, 80, 94, 115, 117, 140, 143, 169, 171, 193, 202, 220, 231, 262, 275, 283]. This type of artifact simply has one large screenshot that captures the entire interface. This artifact has been used most commonly to capture the *visual state* of the application (regardless of the platform), and analyzed further to perform various forms of testing (e.g., regression, acceptance, or test generation).

However, full-interface artifacts capture the visual state of the system at a coarse-grained level of granularity, which renders them less applicable when a more detailed analysis is needed. This is because the full-interface captures the interface as a whole, and is therefore not very useful when the goal is more microscale, such as analyzing or recognizing a particular icon for instance. For these cases, our survey revealed another category containing *localized visual artifacts*. In this case, visual artifacts are created at the level of a specific component, an area of interest, or a certain feature [15, 19, 30, 51, 60, 154, 168, 230]. Compared to full-interface visual artifacts, this type of artifact is more beneficial for scenarios where analysis needs to be performed for a specific component or feature in a system. For instance, localized visual artifacts have been used to create a test case for a GUI (by recording and tracking a single visual artifact for UI elements) [60], or in debugging the rendering or capturing the behaviour of a specific HTML element in a web application [51]. Had the record-and-playback used full-interface instead of localized visual artifacts, then the analysis wouldn't make much since because the test script operates at the level of elements, not entire interfaces.

The third category that emerged is *temporal artifacts*, in which the visual information captures the *dynamic behaviour* of some sequence or chain of information, states, or events [35, 155, 161, 203]. For instance, Bao et al. [35] use a temporal artifact (a video screen recording) to construct a tool to help researchers conducting user studies of developers' behaviours to automatically distill and transcribe their actions, inputs, and event sequences by capturing a video screen recording of their work session.

Finally, the last identified category is *natural input artifacts*. These artifacts capture a natural representation or interaction with a human user. The only example of this artifact that we found in the collected literature are hand-sketches [212, 227]. This type of artifact provides a number of benefits: (1) it provides a more intuitive and natural way for software engineers to interact with, design, develop, or test their software, and (2) it allows a broad degree of freedom in capturing user input, which can be useful when modelling or analyzing multi-variable complex systems.

Visual Techniques Taxonomy.

In this section, we describe a taxonomy of visual techniques. The taxonomy was synthesized from the pool of papers we collected in this survey. This taxonomy has not been used elsewhere, since there are no existing surveys on the use of computer vision techniques in software engineering. For each paper, we analyzed the text and noted down the computer vision algorithms utilized by the authors to solve the software engineering problem being tackled. After conducting this process over all collected papers, we grouped the algorithms and identified three main patterns of algorithms, which are then collected in a taxonomy. We built the

Table 2.5: Major computer vision algorithms used in the collected papers.

Visual Technique	**Algorithm**	**Description**	**Utilized in**
Differential	Image diff	A processing method whose output is a function of the difference between a pair of input images (e.g. PID–Perceptual Image Differencing [279], PHash–Perceptual Hashing [278]).	[34, 35, 51, 62, 66, 78, 79, 80, 140, 143, 155, 156, 158, 168, 169, 171, 185, 186, 203, 230, 275, 286]
	Probability distribution distance	Measuring the distance between *distributions* of pixels in a pair of images (e.g. χ^2 distribution distance).	[32, 64, 66, 94, 115, 117, 161, 186, 220, 275]
Transformational	Color/Spatial transformation	Applying a transformation matrix to the spatial or color-space of one or more images, or transforming spatial regions into abstract data.	[30, 34, 54, 73, 82, 83, 92, 103, 119, 140, 155, 185, 187, 191, 198, 202, 211, 212, 229, 262, 274, 287]
	Optical character recognition	Recognizing images of strings and converting them into textual data.	[19, 36, 60, 193, 203, 252, 274, 281]
Search	Template matching	Finding where an image is located within another image.	[15, 35, 36, 60, 62, 82, 103, 154, 161, 231, 245, 252, 281]
	Machine learning	Finding closest visual matches or categories based on learning patterns in visual data.	[61, 78, 92, 145, 212, 227, 248, 249, 271, 273, 282, 283, 286]

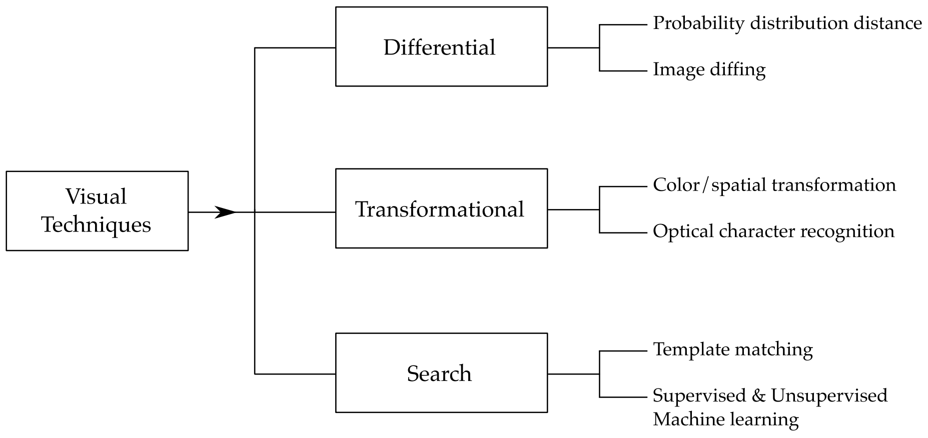

Visual Techniques
Differential
Transformational
Search
Probability distribution distance
Image diffing
Color/spatial transformation
Optical character recognition
Template matching
Supervised & Unsupervised Machine learning

Figure 2.9: Synthesized taxonomy of visual techniques.

taxonomy in an iterative manner, where a new taxonomy category is defined if a certain computer vision algorithm can not be classified under any of the existing categories.

Figure 2.9 shows the resulting taxonomy. As shown in the figure, we identified three main categories of visual techniques used in software engineering research: (1) differential, (2) transformational, and (3) search techniques. Each technique has sub-categories of algorithms, which will be described in Section 2.4.4.

Differential techniques: a visual analysis process that takes two or more visual artifacts as input, and outputs their differences [15, 19, 51, 64, 66, 79, 80, 94, 115, 117, 140, 161, 168, 169, 171, 220, 230, 275]. The nature of these differences is typically case-specific and often targets a specific feature that the paper is considering. Differential techniques are commonly used in situations where a comparison of different aspects, options, or versions of the software is desired. For instance, they have been widely utilized for cross-browser testing [64, 66, 115, 230, 275], where the goal is to check for any differences between two or more browsers in the way they render a web app.

The second category, *transformational techniques*, achieve a specific software engineering task by transforming the visual artifact into a more abstract type of information [30, 35, 78, 143, 193, 202, 203, 212, 227, 231, 262, 283]. This transformation is typically case-specific, as the higher-level abstract information is used to solve the specific instance of problems addressed in the paper. For instance, this approach has been used to allow manual hand-drawn strokes as a method to specify test executions and requirements [227], where transformational techniques are applied on hand-drawn stroke instances to extract testing instructions and assertions from the strokes, and finally generate a working test case.

Finally, the third category is *search techniques* [60, 62, 154, 155]. In this case, a visual artifact is used as a key to find information within a larger set of visual artifacts. A popular example of this approach is visual record-and-playback tools such as Sikuli [60]. These tools first record component visual artifacts for every GUI element clicked or interacted with by a developer or user, and then, in the playback phase, a visual search method is employed to locate the element on screen to perform the recorded action (e.g., click).

In order to have a better insight on the use of different categories of visual techniques, the bubble plot of Figure 2.10 shows their distribution over the various SE areas. In the plot, software testing has a finer-grained granularity since it is by far the most represented area in our final pool of papers. We make a number of observations from the plot. First, we notice that transformational techniques are more uniformly represented across all SE areas and tasks. This is a somewhat expected result since transformational techniques are generic enough to be used

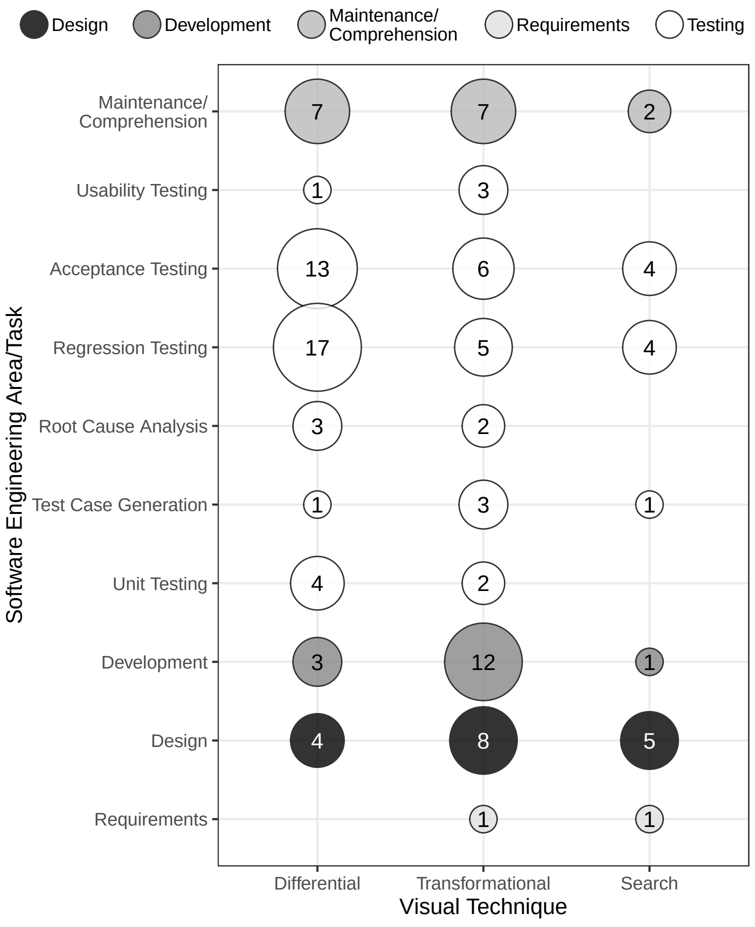


Figure 2.10: Distribution of visual techniques per SE area & task.

on their own, or in addition to other techniques as pre- or post-processing steps. Next, we also notice that differential techniques were relatively common in testing tasks. For instance, regression and acceptance testing greatly benefit from differential techniques, since they are very well aligned towards detecting differences and therefore suitable for indicating regression faults.

Computer Vision Algorithms

In addition to the preceding analysis of the high-level categories of visual techniques, we also examine the specific computer vision algorithms used in the collected pool of papers. Table 2.5 shows the algorithms that have emerged from our analysis. We now describe each of the identified algorithms.

Image Diff. In these CV algorithms, a pair of images is taken as input and the output is defined as a function of the *difference* between the pair. Various instances of these algorithms differ in their choice of the output function. For instance, the most common variation uses the raw absolute difference as the output value [35, 62, 66, 79, 140, 155, 168, 185, 203], where it is useful for faithfully detecting any *pixel-level* difference. Another variation adopts a more relaxed approach where the output function captures only *perceivable* differences by humans using algorithms such as PID (Perceptual Image Differencing) [279], PHash (Perceptual Hashing) [278], and Structural Similarity Index (SSIM) [264], which aim to reduce false positives by mimicking human perception. These techniques were utilized in [51, 169, 171, 185, 186, 275].

Probability Distribution Distance. These algorithms quantify distances in *populations* of pixels, as opposed to a pixel-wise comparison. The goal here is to measure how similar two given distributions are, such as image *histograms* [197] which give the distribution of pixels in an image. This is then used to establish whether the two distributions of pixels can be assumed to represent similar visual information. The χ^2 histogram distance (for both coloured and grayscale data) is by far the most commonly used distribution distance in our pool of papers [64, 94, 115, 117, 186, 220], as it is readily available in many implementations and provides a simple and effective approach for the needed quantification of distance.

Color/Spatial Transformation. These algorithms perform a transformation of the spatial or color-space of one or more images [30, 140, 155, 185, 187, 202, 262, 274]. This constitutes applying a 2D or 3D transformation matrix on the desired geometric space (e.g., a rearrangement of color-space). This class of algorithms has been used in our pool of papers to perform tasks such as extracting structure from images, aligning images, and various forms of thresholding to extract content.

Optical Character Recognition (OCR). OCR algorithms use a series of computer vision analyses to recognize strings in images. Once the string is recognized, another sequence of steps converts each character in the image to textual data. We found multiple papers in our pool (e.g., [19, 36, 60, 193, 203, 252, 274]) which have used OCR for tasks such as checking GUI component labels and generating component labels from mockups/screenshots.

Template Matching. Another major class of CV algorithms used in the papers is template matching. Here, one image is searched within another image or set of images. That is, a visual scan is performed to find a template image (hence the name) in a larger image or set of images. Several papers in our pool [35, 60, 62, 154, 161, 231, 245, 252] have used this approach to achieve tasks such as locating and finding coordinates of components and checking the presence/absence of components or certain features within a set of GUIs. While it may appear that template matching is related to the OCR class of techniques, it is actually different. This is because OCR converts an image to a string, while template matching finds the location of a given image within another image.

Machine Learning. Machine learning has also been used by a few papers in our pool. For instance, *decision trees* were used for classifying web pages in cross-browser testing [64, 231]. *Convolutional neural networks* (CNNs) were also used to analyze GUIs and their content [187, 231]. Furthermore, scale- and transformation-invariant features (e.g., SURF–Speeded-Up Robust Features [38], Wavelets [75]), which are basically features aiming at analyzing image structure and content, were used to classify and detect GUI elements [35, 143, 245, 274].

Libraries and Tools.

We also investigated what industrial or open-source libraries and tools were used by the papers in their implementation of computer vision techniques. Table 2.6 shows a list of the CV tools or libraries that have been used by the papers in our pool. The first column reports the name of the tool or library. The second column describes the scope of the library, and the last column shows the paper(s) that utilize a certain library to implement their CV analysis or processing. Papers that do not state which library or tool they used are not included in the table, and therefore the number of papers in the table is smaller than the total pool. For the sake of completeness, we also included other CV tools and libraries that have not been used by any of the papers in our pool, which are marked by the "N/A" (not applicable) value in the last column. These tools were included in order to present the software engineering research community with an easy access to a list of other computer vision tools that they might find helpful to use. Due to the absence of queryable databases in which computer vision tools are listed, we resorted to manual search engine queries to find computer vision tools or libraries other than those already used by our pool. The queries we used were of the form "alternative (libraries or tools) to X", where X is one of the libraries already in our pool.

Table 2.6: Open-source and industrial computer vision tools and libraries utilized by papers in the pool.

Name	**Scope**	**Utilized in**
OpenCV	provides data structures and algorithms for a wide variety of advanced computer vision processing	[34, 35, 60, 64, 66, 94, 103, 154, 161, 169, 171, 187, 193, 220, 245, 252, 274, 286]
Tesseract	extensible and modular open-source optical character recognition (OCR) engine	[187, 193, 203, 281]
FineReader	comprehensive OCR engine that includes relevant pre/post-processing steps and formats	[36]
BoofCV	performance-oriented library that focuses on real-time processing	[203]
ImageMagick	simple and easy to use tool and library for basic image transformations and analysis of color	[168]
ITK (Insight ToolKit)	specialized in image and coordinates matching, with comprehensive support for high-dimensional data	N/A
VLFeat	geared towards feature extraction, covering a large set of feature descriptors	N/A
ImageJ	a modular tool and library with an extensive number of plugins for various image analysis tasks	N/A
Amazon Rekognition, Google Vision AI, Azure CV	cloud-based tools with a large number of pretrained analysis and recognition models	N/A

The most commonly used library is *OpenCV*. [1] This library has implementations for a large number of CV algorithms, including all the algorithms we report in this survey (Section 2.4.4). It was first released in 2000, and is still in active development. Other than OpenCV, a few other libraries were used by only a handful of papers. *ImageMagick*[2] is a simple and easy to use library offering basic quick image manipulations (e.g., resize, rotate). *ImageJ*[3] is also quite similar in features and scope. *Tesseract*[4] is a popular open-source library that is modular and extensible, with support for more than 100 languages. *Abby FineReader*[5] is a commercial library that focuses specifically on OCR and includes many related preprocessing/postprocessing algorithms, as well as various file formats. *BoofCV*[6] focuses on performance-tuned algorithms and is geared towards cases where real-time response is priority. *ITK*[7] focuses on high-dimensional visual data often present statistics and science, and also has extensive image coordinates matching algorithms that might be beneficial in a number of image matching applications. *VLFeat*[8] is geared towards implementing a wide variety of feature extraction and matching algorithms, enabling applications in image search or transformation. Finally, there are cloud-based tools offered by major cloud hosting providers (e.g. Amazon Web Services, Google Cloud Platform, Microsoft Azure). The defining feature of these services is their cloud nature and the availability of many pre-

[1]https://opencv.org
[2]https://imagemagick.org/
[3]https://imagej.net
[4]https://github.com/tesseract-ocr/tesseract
[5]https://abbyy.com/
[6]http://boofcv.org
[7]https://itk.org/
[8]https://vlfeat.org/

Table 2.7: Summary of the analysis approaches for the targeted research problems.

Non-functional Property	Research Problems	Approach
Accessibility	- testing semantic roles - repairing form labels	- identifying page regions, then computing and sorting ratios of visual locations, areas, visual size variance, and neural network classification. - abstracting form elements, then combining visual layout (relative spacing) with visual prominence (weight, size, color), and finally optimizing label assignment
Maintainability	- generating reusable components	- visual normalization of the page based on element location, dimensions, and type, followed by histogram clustering
Testability	- testing canvas elements	- detecting and isolating object boundaries, then identifying their shapes, dimensions, colors, and hierarchy

trained computer vision models for various visual tasks, such as content tagging and visual path analysis.

Overview of the proposed techniques. In this dissertation, we propose a number of visual analysis techniques to improve the non-functional properties of testability, accessibility, and maintainability. An introduction to these properties and the motivation for exploring them is discussed in Chapter 1. In Table 2.7, we show an overview of the approaches used, which are presented here only in a high-level and will be discussed in detail in their relevant chapters. While existing works are mostly based on using entire images, say, for diffing or searching, the techniques in this dissertation operate at a fine-grained level of analysis. By fine-grained we mean that the analysis focuses on very specific visual features that may potentially be helpful in addressing the research problem. For instance, when we analyze canvas elements, one fine-grained aspect that is used is extracting and identifying the boundaries of each object, in order to help detect the type of each object and express it in the DOM. Accordingly, the approaches are based on identifying what fine-grained features should be selected, where should they be collected from, how would they be combined, and how to analyze them to address the research problem. Each of these aspects is problem specific and need to be explored and determined according to the objective and constraints of the problem.

Summary. The goal of this subsection was to explore what computer vision techniques were used and what visual artifacts were extracted from the software. To this end, we identified four categories of artifacts. The first category, full-interface artifacts, represents cases where the entire visual content of a software is used (e.g., entire desktop interface, entire console). The second category is localized artifacts, where only a specific module or component of the software is captured visually. Third, temporal artifacts capture the dynamic behavior of the states or events in a software. Finally, natural input artifacts capture natural forms of input by humans (e.g., hand sketches). Visual artifacts are then processed by one or more of three visual techniques. The first category, differential techniques, are based on various forms of contrasting two or more visual artifacts and using the differences to solve a software engineering problem. The second category, transformational techniques, rely on transforming the visual artifact into a more abstract data structure on which further analysis can be conducted. Finally, in search techniques, a visual artifact is used as a key to find information within a larger set of visual artifacts.

Table 2.5 shows the major computer vision algorithms used in the collected papers. The visual technique column describes the high-level goal of what the algorithm is trying to achieve visually. The algorithm column lists the specific algorithms that were used to achieve the task. For instance, when papers wanted to do a differential examination of visual artifacts, the two computer vision approaches that were used were image diffing and distribution distances.

2.4.5 Evaluation and Challenges (RQ4)

Evaluation Methods.

We systematically studied the selected papers to understand the most common methods used for evaluating the proposed visual approaches. Table 2.8 (Evaluation Methods) depicts the results: the evaluation methods presented are not necessarily disjoint, since one technique can for instance be evaluated with respect to its performance (i.e., running time), and its accuracy calculated by the judgment of human participants.

In more than half of the works (53%), the evaluation methods were performed using wide-spread effectiveness assessment measures such as precision and recall [30, 35, 62, 64, 66, 94, 115, 117, 140, 143, 161, 169, 193, 202, 212, 220, 262, 275].

Another significant body of work measured the manual effort saved by the visual approach with respect to the humanly performed task [62, 80, 94, 154, 155, 155, 169, 212, 231]. Moreover, several works also evaluated the performance of the proposed approaches, usually measuring the running time on common developer platforms (i.e., mid-level notebooks) [35, 117, 140, 193, 202, 212, 230, 262].

Performance is potentially considered as important as the accuracy because image analysis techniques are deemed as being computationally expensive. Thus, their application and use must carefully evaluate the overhead imposed by these visual approaches over the most general SE tool being developed. Initially, this might not favor their adoption if compared to other less computationally-intensive approaches. However, the authors report that the time taken by their proposed techniques are all reasonable in common processing environments, suggesting that execution time should not be considered as a deterrent decision criterion when adopting a visual approach to support a SE task.

In several works, the human expertise was used to assess the output of the visual techniques [64, 66, 117, 143, 169, 171, 203, 220, 275]. As an example, Mahajan and Halfond [169] examine the efficacy of WebSee (a tool that identifies presentational failures in HTML pages) through manual investigation of its output when it is executed on 253 automatically-generated test cases. This is done to see whether WebSee is able to correctly identify the faulty HTML elements seeded when generating the test cases. While we were expecting more human judgment involved in the evaluation of the selected papers, its application depends largely on the context and the problem being solved.

Four works do not propose a quantitative empirical evaluation of the proposed technique, but rather present them by means of use cases. A closer look at these publications revealed that two of them are short papers [15, 227], where having a more thorough evaluation is not mandatory. The other two papers [51, 78] were published at the ACM Symposium on User Interface Software and Technology (UIST), in which use case demonstrations appear to be more frequent. Four works included an industrial evaluation [19, 143, 155, 169] where the work is evaluated through feedback or measurements in an industrial setting.

Finally, we noticed diversity in the evaluation methods of techniques aimed at solving similar problems. For example, several works have been proposed to identify some sort of *visual defects* (e.g., all the approaches dealing with XBI). Four of these techniques [168, 169, 171, 202] use “seeded faults” as an approach for constructing test oracles, whereas others take a different evaluation path. For instance,

Table 2.8: Evaluation methods and challenges.

Evaluation Methods	
Accuracy Measures	[30, 32, 35, 61, 62, 64, 66, 92, 94, 115, 117, 119, 140, 143, 161, 169, 185, 186, 187, 191, 193, 198, 202, 212, 220, 245, 248, 249, 252, 262, 273, 274, 275, 286]
Comparison against manual work	[32, 62, 80, 94, 154, 155, 155, 156, 169, 185, 186, 187, 212, 231, 252, 282, 287]
Survey or manual validation of results	[36, 54, 61, 64, 66, 117, 119, 143, 156, 158, 169, 171, 203, 220, 271, 273, 275, 281]
Time comparison	[35, 36, 54, 61, 92, 117, 140, 191, 193, 202, 212, 230, 245, 252, 262]
Comparison against competitor approaches	[15, 54, 73, 79, 115, 161, 171, 245, 275, 283]
Comparison to original design/output	[30, 154, 193, 202, 262]
Use case demonstration	[15, 34, 51, 78, 103, 227]
Industry context/feedback	[19, 36, 73, 119, 143, 155, 169]
Comparison with random/brute force	[94, 168, 202, 230]
Seeded faults	[168, 169, 171, 202]
Differential comparisons	[60, 171]
Line/state coverage	[79, 283]
Comparison on different physical devices	[161, 211]
Challenges	
Noise	[34, 35, 54, 73, 92, 119, 140, 143, 154, 155, 161, 185, 186, 187, 191, 193, 203, 211, 212, 245, 252]
Dynamicity	[30, 36, 60, 66, 156, 158, 202, 262, 287]
Recognition	[34, 35, 36, 54, 61, 73, 92, 103, 119, 140, 143, 185, 186, 187, 191, 198, 227, 245, 248, 249, 252, 271, 273, 274, 281, 282, 283, 286]
Visibility	[35, 36, 60, 66, 154, 158, 283]

Roy Choudhary et al. [220] manually count all XBIs detected by different tools (i.e., their agreement) as an upper bound for the number of issues that an XBI detection tool should identify, and use this number to compute the recall of X-PERT. As a further evaluation, authors could have been also evaluating their technique using seeded XBIs to broaden the evaluation. Conversely, the works that only use seeded faults could use the agreement of multiple fault detectors to compute recall. This scenario essentially suggests that the evaluation of some of the proposed techniques in the literature can be substantially improved by leveraging the evaluation approaches from other techniques that aim at solving similar problems.

Challenges and Limitations.

Evaluations also exposed the main challenges and limitations that authors faced while developing their visual-based solutions. Unfortunately, a subset of the papers (41%) either do not explicitly discuss drawbacks for the visual approach being evaluated, or do not provide concrete failing examples. On the other hand, the majority of the papers (59%) do report the challenges encountered during the experimentations, summarized in Table 2.8 (Challenges), which we describe next.

Noise. The most recurring technical issue that inhibited the correct functioning of visual approaches concerns the *noise* present in the visual artifacts [35, 143, 154, 155, 161, 193, 203, 212]. Noise refers to the presence of other random or overlapping items in the background of the visual artifact that prevents, for instance, the correct detection of the target object. Hence, before applying the desired visual method, a pre-processing technique is often used to clear the noise out of the artifact.

As an example, Ponzanelli et al. [203] apply OCR to detect source code in frames sampled from video tutorials. However, the effectiveness of OCR fluctuates dramatically if the considered frame's background contains non-pertinent information. As a solution, the authors utilize two additional visual techniques—shape detection and frame segmentation—in order to focus OCR towards the area of interest.

Another more specific noise-related limitation pertains to the *sensitivity* of the visual approach to theme/surroundings changes, lighting conditions, and reflections [60, 140, 143, 154, 155, 161]. A possible mitigation concerns using threshold parameters to limit their detrimental effect.

Dynamicity. Highly-variable visual artifacts are also a major challenge for the design and development of reliable visual approaches.

One source of fragility is due to animations [30, 60, 66]. Indeed, a continuously-changing visual artifact (e.g., a video frame) represents a major limitation for many one-shot techniques. As a possible solution, such techniques would need to be applied repeatedly, similarly as real-time domains, or, in extreme cases, re-designed from scratch to meet the new domain requirements.

Another source of fragility is due to web advertisements [262]. An ad is rendered as a dynamic component within a more static container (i.e., a web page), the visual representation of which usually changes across consecutive loads of the web page or over different time stamps. Visual methods need to isolate the dynamics of web pages to avoid erroneous behaviour or false positives.

Recognition. Another challenge of visual approaches that emerged from our study concerns (1) accurately identifying small imperceptible differences between images and (2) recognizing complex user-defined actions such as manually inserted strokes or handwriting.

For example, a source of false positives in VISOR [140], an image comparison technique used for black-box testing of digital TVs, is the tiny differences that occur when rendering accent characters on the screen (e.g., cedillas). Typically, the choice on whether such differences should be considered as defects or not is left to human's perception. Two other works experienced recognition issues, but not at the same pixel-level as VISOR. Bao et al. [35] discuss problems in detecting visual fine-grained developers' actions from videos, such as code editing, text selection, and window scrolling. Similarly, Scharf and Amma [227] mention issues in detecting handwriting in manually-produced sketches.

These findings demonstrate how complex is the set of visual differences that can emerge while comparing two images, and how vast and multifaceted is the set of input actions that are possible on a user interface. Indeed, when a visual method is used to actively support the *end user*, they arguably need to recognize a broader set of inputs than those they would receive from another algorithm, if they were executed in a software controlled environment. This poses new challenges to the development of robust visual approaches for aiding SE.

Visibility. At last, in four works, having the visual artifact present in the main rendering area is mentioned as a requirement for the visual technique to effectively fulfill its task. This requirement can be violated by, e.g., *fading* [60, 154] and *scrolling* [35, 66].

As an example, Leotta et al. [154]'s tool PESTO uses template matching to detect the optimal visual locator corresponding to a web element on the page. However, the web element of interest might be outside the actual rendered web page. This happens when a long and complex form with multiple fields cannot be entirely visualized within the main screen. The authors propose an engineering workaround to solve issues related to scrolling: if the visual artifact being searched is not immediately displayed, the tool scrolls the page down automatically and the template matching is repeated.

2.5 Discussion

Increasing Adoption of Visual Approaches. Our findings show a general growth trend in the adoption and use of visual approaches in the SE community. Most of the surveyed papers explicitly recognize, and empirically demonstrate, the contribution brought by visual methods in supporting SE tasks.

Based on our examination of the literature, we attribute this increase to two factors. First, many software developed nowadays have a GUI or other visual interfaces. The end-user experience is increasingly becoming more important in adding value to software, and therefore the adoption of visual methods is expected to increase further in the next years. Our examination of the trend of number of annual publications already shows this trend of increasing number of works utilizing visual techniques. Second, the rapid pace of improvement in hardware and processor architecture has made the efficiency and run time of advanced visual techniques feasible in common development environments, which we expect will cause an increased adoption of visual approaches further down the line.

Software Testing a Major Driver of Visual Approaches. The majority of papers (around 75%) have focused on the research area of software testing. Our intuition behind this is two-fold. First, testing is one of the most active SE research areas in general, so it is not surprising that most of the collected papers fall within this category. Second, testing is largely a tooling-based research area, in which tool prototypes are developed and empirically evaluated. Many different static and dynamic analysis tools are proposed each year to facilitate test engineers' activities. The results of this survey show that the visual perspective of the software has been recently used to complement static and dynamic analysis because it provides a novel and complimentary perspective of the software under test. The types of analyses that are performed on the presentation-level of the application would be likely very difficult to perform by analyzing the source code only, especially with the increasingly complex interfaces and the great emphasis placed on user experience, of which the interface is a cornerstone component.

Custom Solutions. The visual techniques used in the collected papers are often adhoc solutions developed for tackling a specific problem. Authors recognize that it is unlikely to have a consolidated and broadly-accepted solution. More specifically, all collected papers have discussed, to some extent, the need of visual approaches for parameter fine-tuning, such as optimal threshold selections. In contrast, as will be discussed in the next chapters, the techniques proposed in this dissertation aim to minimize or eliminate the need for parameters or thresholds whenever possible.

Manipulating visual artifacts through a visual technique is highly application-specific, both in the adopted approach and in the considered domain [277]. For instance, this survey highlights a large body of work in the area of cross-browser incompatibility. The authors of these papers have adopted a large variety of solutions (or incremental variations) to tackle the same problem. To mention a few, Choudhary et al. [66] use an image comparison measure based on Earth Mover's Distance (EMD), whereas Mahajan and Halfond [169] adopt perceptual differencing, and He et al. [115] compare the colour histograms, among other approaches. This trend can be partially explained by the need of proposing and experimenting with novel and potentially useful techniques. However, a researcher approaching this topic for the first time could be somewhat disoriented. In fact, given that the solutions for the same problem are many, and they are often evaluated on different benchmarks, it is not straightforward to find an agreement on what the best technique could be. This led to a landscape where each work would typically experiment with a custom visual processing pipeline to address the specifics of the SE task at hand.

Need for Visual Benchmarks in SE. We highlight the lack of comparative visual benchmarks on which to evaluate the plethora of visual approaches utilized in software engineering research. A repository of standard, well-organized, categorized, and labeled visual artifacts could be very useful to support empirical experiments, and to guide the next generation of research utilizing visual approaches for software engineering tasks. Such repositories exist in traditional (non-visual) software engineering research, such as SIR [84], Defects4J [132], SF100 [98], and BugsJS [109, 110]. This has not been the case, however, for visual techniques in software engineering. For instance, having analogous repositories for visual bugs can foster further applications of visual methods in software testing. In the issues discussed in this dissertation, which are all non-functional properties, no visual benchmarks exist either, since the field is still at its early stages and hasn't matured yet to the degree of creating benchmarks.

Similarly, object detection and classification tasks need labeled images. In computer vision literature, there exist some pre-validated and labeled visual benchmarks, such as ImageNet [221], BSDS500 [173], or Caltech 101 [93]. In software engineering, a benchmark of labeled visual artifacts might aid in developing visual techniques, or training systems for machine learning and deep learning scenarios. A notable step in this direction has been carried out in the Rico [79] repository. The repository contains around 70k labeled UI screenshots, each of which are labeled with visual, textual, structural, and interaction trace data. The dataset facilitates software engineering tasks related to the UI, such as UI design search, UI layout generation, and UI code generation.

Maintainability of Visual Artifacts. The visual artifacts created or extracted from the software are rarely static across time, especially for rapidly evolving software such as in agile environments. The artifacts would therefore have to be frequently modified or updated to keep track of the underlying evolving software. Alégroth et al. [15] indicate that the maintainability of visual artifacts produced and used by the visual testing tools as being a major challenge of the visual-based testing approaches. Potential research directions to mitigate this challenge include proposing strategies for conducting cost-benefit analysis depending on the expected degree of visual evolution of the software, and devising automated techniques to help with or reduce the maintainability effort for visual artifacts.

Familiarity with Computer Vision. Perhaps the biggest challenge hindering a wider adoption of visual approaches in software engineering is the lack of familiarity with computer vision techniques. For instance, Delamaro et al. [80] describe how developers should have basic knowledge of image processing in order to even *use* the proposed tool in the paper. This is because the visual artifacts can structurally vary with each use, and thus sometimes one or more manual image processing adjustments need to be performed before being able to process the visual artifacts. In the work proposed in this dissertation, we mitigate this issue by proposing techniques that reduce or eliminate the need for parameters or adjustment thresholds, in order to improve robustness.

Threats to Validity. The threats to validity of this survey are the bias in the papers' selection and misclassification of the pool of papers in the various research questions. We mitigate these threats as follows. Our paper selection was driven by the keywords related to visual approaches and software engineering (see Section 2.3.4). We may have missed studies that use visual methods in the software engineering activities that are not captured by our terms list. To mitigate this threat, we performed an issue-by-issue manual search of the major software engineering conferences and journals, and followed through with a snowballing process. Concerning the papers' classification, we manually classified all selected papers into different categories based on the targeted SE area, as well as, more fine-grained sub-categories based on their domains, tasks, and the utilized visual methods (see Section 2.4). Identifying the rationale from the papers that do not explicitly mentioned it involved some subjectivity and may have resulted in suboptimal mappings, which constitutes another threat. However, there is no ground-truth labeling for such classification. To minimize classification errors, the first three authors of this chapter have carefully analyzed the full text and performed the classifications individually. Any disagreements were resolved by further discussion.

2.6 Conclusions

A recent and growing trend in software engineering research is to adopt a *visual perspective* of the software, which entails extracting and processing *visual artifacts* relevant to the software being analyzed. To gain a better understanding of this trend, in this chapter, we surveyed the literature on the use of visual analysis approaches in software engineering. From more than 2,716 publications, we systematically obtained 66 papers and analyzed them according to a number of research dimensions. Our study revealed that visual analysis techniques have been utilized in all areas of software engineering, albeit more prevalently in the software testing field. We also discussed why visual analysis was utilized, how these techniques are evaluated, and what limitations they bear. Our suggestions for future work include the development of common frameworks and visual benchmarks to collect and evaluate the state-of-the-art techniques, to avoid relying on ad-hoc solutions. We believe that the findings of this work illustrate the potential of visual approaches in software engineering, and may help newcomers to the field in better understanding the research landscape.

A number of key findings can be observed from the survey in order to help in guiding and framing the remainder of the dissertation. First, cross-browser testing is by far the most common area of application, whereas exploring non-functional properties received little to no focus. This shows that there is an opportunity to explore the use of visual analysis in improving non-functional properties, and therefore the remainder of the dissertation will be focusing on this aspect. This will provide better and more novel research contributions compared to exploring functional properties. Furthermore, our survey of the visual analysis techniques used shows that the majority of existing works use some form of basic image diffing or features. This shows that it would be novel and potentially useful to investigate more fine-grained level of visual analysis that examines fine-grained visual details. Accordingly, this unexplored approach will be the basis of the techniques proposed in the remainder of the dissertation.

Chapter 3

Web canvas testing

3.1 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 [21], geographical modelling [67], and genetics research [50]. The `<canvas>` element on the HTML page is only a container for graphics, which is updated programmatically, via its APIs, through JavaScript code.

From a testing perspective, however, there has been little to no progress in the literature in terms of testing canvas elements. None of the works surveyed in Chapter 2 perform canvas testing. In addition, existing web testing methodologies do not apply to canvas elements for the following reasons. The testing of web applications in practice is based on using a browser automation tool, such as Selenium[1] or PhantomJS[2], to make assertions on the DOM (Document Object Model) of the

[1]http://www.seleniumhq.org/
[2]http://www.phantomjs.org/

webpage, which is a tree representation of the current set of nodes in a page. 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 cannot be used with canvas elements. These elements only expose 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. All 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 (e.g., asserting objects properties or locations) is not fully supported in these tools. More specifically, a visual test consists of a image comparison, which does not provide information regarding the objects on the canvas,

[3]http://www.sikuli.org/

[4]https://www.testplant.com/

such as their shape, location, or color, etc; (3) from a maintenance viewpoint, the commonly used visual testing approach of image diffing has been shown to be more fragile compared to DOM-based methods [70, 147], because changing even a slight change in one pixel in the image would cause a false positive (i.e., false test failure). 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.

In this chapter, 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. The first approach for making web canvas elements testable 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.

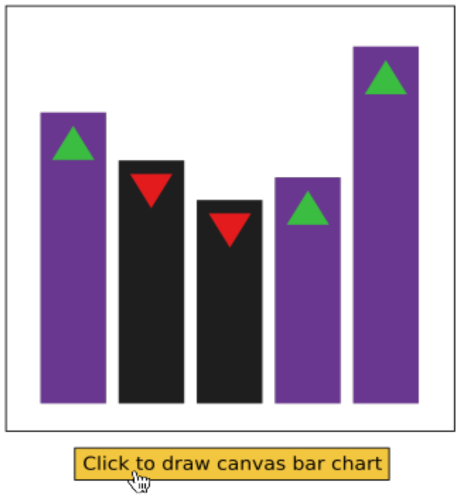


Figure 3.1: An example canvas-based application. When the button is clicked, the plot is drawn dynamically (on the client side) using the canvas element.

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.

3.2 Motivating Example

Figure 3.1 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 3.1. 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 3.1 shows a snippet of how the setup and manipulation of canvas elements is done exclusively through the Canvas JavaScript API [5]. 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 3.1, 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 [5] of canvas elements, the canvas has no DOM representation. That is, the DOM subtree under a canvas node remains empty, and therefore its state remains unobservable. Instead, the canvas API functions are required to draw *directly* to the raw monitor pixels buffer without the costly task of maintaining a DOM representation. This direct drawing allows canvas elements to achieve high-speed performance in order to enable dynamic and highly-interactive applications.

Unfortunately, this DOM-free nature 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 unobservable. 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 chapter, we propose an approach that makes canvas elements testable. That is, the goal is to provide developers with the fundamental capability of observing the canvas state and making assertions on it. However, the aim is not to provide a complete testing solution, but rather to enable the testing process itself, thereby improving testability. 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 techniques. Finally, the developers would write their own tests, or could optionally use the automatically generated tests to check the visual objects of the canvas and their properties.

3.3 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 ana-

```

function onButtonClick() {
  var canvas = document.getElementById("canvasBarChart
").getContext("2d");
  ...
  canvas.beginPath();
  ...
  // Example: this would draw one of the
  // rectangles in the bar chart.
  canvas.fillRect(leftCoordinate, topCoordinate,
          width, height);
  ...
}


// The HTML portion of the app
<canvas id="canvasBarChart">
  // This tag remains empty throughout the usage of
  // the app, due to the lack of DOM representation
  // for canvas elements.
</canvas>
```

Listing 3.1: JavaScript and HTML snippet of the canvas drawing in Figure 3.1.

lyze canvas elements contained in a larger webpage or canvas applications that are solely composed of a single canvas element.

3.3.1 Overview

Figure 3.2 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 3.1. Subsequently, the approach builds a

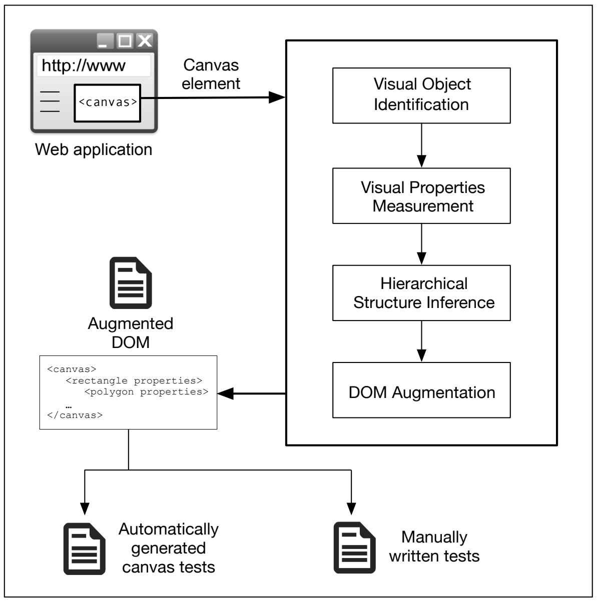


Figure 3.2: Overview of the proposed approach.

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.

3.3.2 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 [254], namely HSV, L*a*b*, L*u*v*, CIE-RGB, XYZ, YUV, YIG, YPbPr, and YCbCr. Each of these is simply a mathematical formula [254] 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 3.3. The figure shows how much contrast there is in the canvas image when using the different color conversions. The contrast is measured using the variance in pixel values. Higher values in the figure 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, $\mathbf{\bar{C}_T}$, which represents the canvas

[5] http://corehtml5canvas.com

screenshot in YCbCr. An example of the outcome of this step is shown in Figure 3.4-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 [95]. 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 [228]. We make this choice because this method has been shown [142] 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 $\nabla\bar{\mathbf{C}}_{\mathbf{T}}$.

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

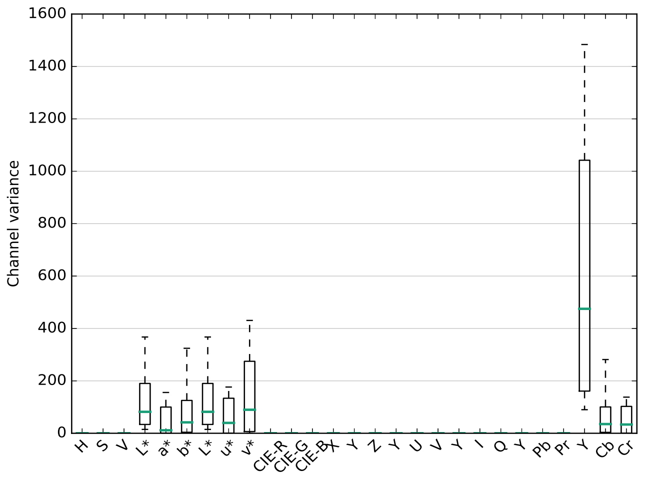


Figure 3.3: Parameter estimation for the color contrast adjustment step of the algorithm. Choosing the YCbCr color representation yields best contrast adjustment for the canvas screenshot. Higher values are better.

been shown [234] to yield high performance. The outcome of this step is depicted in Figure 3.4-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 $\mathbf{B}_i$:

$$(\nabla \bar{\mathbf{C}}_{\mathbf{T}})^B = \bigcup \mathbf{B}_i \ : \ \bigcap \mathbf{B}_i \equiv \emptyset \tag{3.1}$$

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* [241] 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 $\mathscr{E}$ that was just computed in order to check for branching in a boundary. A value of $\mathscr{E} = 0$ 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 $\mathscr{E} < 0$ indicates that a boundary has branching. In this case, the algorithm breaks down the original boundary and creates new boundaries for each branching. We note that this will not cause any unnecessary or excessive fragmentation of the structure, because the fragmentation only occurs when the boundary branches, and no branching occurs for continuous (i.e., unbranched) segments of the object.

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 3.4-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. While more complicated computer graphics models might be used (e.g., curves, polynomials), we opted for linear segments to simplify the analysis and make it faster.

Following the detection of all boundaries $\mathbf{B}_i$, we need to understand what the *identity* of each boundary is. This is because, at this stage, we only have a collection of boundaries but do not know what shapes do they represent, or what their properties are (e.g., dimensions, colors). The first step towards gaining this information is to segment these boundaries or break them down into smaller sections. Accordingly, the analysis proceeds by populating each boundary with a random set of probabilistic Hough segments from the computer vision literature [175]. These segments are small linear portions of the boundary of each region. This approach has been shown [137] 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 is because the aforementioned Hough segmentation process is stochastic in nature and while it is robust in most situations, its stochastic nature does cause to generate false positives. 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 [111] 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 (which represent the bounding boxes of objects) 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 be-

Table 3.1: 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)	- centroid: geometric center point of objects - size: - width and height: for rectangles and squares - diameter: for circles	- z-order: relative order of objects in the z-direction (i.e. behind or in front of other objects)
* Note: our approach allows multiple overlapping, partially visible, or nested combinations of these objects.	- color (HTML hex values) - orientation: number of degrees of rotation from horizon - point-set: the set of points representing generic shapes (polygons)	- Parent-child relation: which objects are contained inside other objects.

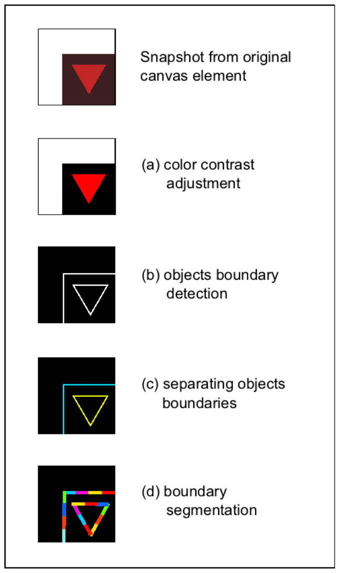

Snapshot from original canvas element
(a) color contrast adjustment
(b) objects boundary detection
(c) separating objects boundaries
(d) boundary segmentation

Figure 3.4: Illustration of the general stages involved in the visual identification of canvas objects. Best viewed on a color monitor.

longing to the 'line' type. Similarly, a set of three connected 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*. These set of conditions define what a circle is exactly. This is because 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 3.1 lists the different object types recognizable by our approach. These types are the same types of objects provided by the canvas API. We note that our approach does not require any training dataset like machine learning approaches, because the attributes and identities of various shapes can be more precisely and robustly captured using mathematical ratios such as the process conducted in this step.

3.3.3 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 3.1 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.

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 average 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 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 because the ellipse's major and minor axes would align with the rectangle's axes. 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 3.1. 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 for the computer vision literature [222], which is a mathematical transformation that detects points of intersections in boundaries. This process extracts points with more than one directions of lines coming in/out of it, which yields points where linear segments intersect.

3.3.4 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 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, **P**, captures parent-child information. The second graph, **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 **P** and **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 **P** and **Z**. The update to **P** is a new directed edge from the child to the parent. Similarly, the update to **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 **Z** from the object with the concave parts to the object that is resulting from the R-tree query.

3.3.5 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 3.1 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 **P** graph, and the z-order relations contained in the **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 3.2 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 3.1, 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,

```
<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 3.2: The visually inferred augmented DOM for the canvas element of the motivating example (Figure 3.1 and Listing 3.1)

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 3.3. 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. However, we emphasize that the goal of the proposed approach is to provide developers with the *ability* to test canvas elements, since they are not testable at the moment. That goal is the priority, more so than providing a complete end to end testing solution that would cover all the possible ways or markups that developers would prefer to write their tests in.

Implementation. We implemented our canvas visual inference approach in a tool called CANVASURE [57]. CANVASURE is implemented in Python 3. We use the numpy [261] library to import basic mathematical and numerical functions, and use the scipy [127] library for matrix computations on the screenshots. We use the Selenium web driver to run and instrument web applications and extract canvas elements.

3.4 Evaluation

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

```
// 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 3.3: The automatically generated test assertions for the canvas element of the motivating example (Figure 3.1 and Listing 3.1)

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

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

3.4.1 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. The rationale for this criterion is that we found some instances were the subject was not a canvas-based application, but rather only used small (e.g., icon-sized) canvas elements to display icons, and therefore this does not represent a canvas element that is rich and complex enough to be fairly included in the evaluation. We note that our approach is 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 five open-source applications that use canvas elements. A list of these applications is shown in Table 3.2. 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.

3.4.2 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 can-

Table 3.2: List of web applications used for evaluation

Application	Description	LOC
JBrowse [267]	Genome browsing and visualization	105,427
Reactome [74]	Reactions analysis	27,702
Scribl [182]	Genetics analysis	20,939
Gibberish [214]	Music composition and production	14,884
iCanplot [236]	Data plotting and visualization	7,269

vas 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 by rendering an image from the structure and properties in the augmented canvas DOM. Finally, we compare the similarity between the snapshot of the original 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:

$$\Delta E = 1 - \frac{\sqrt{\frac{1}{n}\sum (C_{DOM} - C_O)^2}}{\| (C_{DOM} - C_O) \|_2} \tag{3.2}$$

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, from a more practical point of view, we do not have access to the objects on the canvas due to the lack of observable state in the canvas, which is the very problem that our approach is trying to solve. Accordingly, due to absence of access to canvas objects, simulating a fault of removing a certain object is not practically achievable. Finally, 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 accuracy in 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 [268] 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 two reasons. First, the DOM is the source against which any assertions are made, whether automatically or manually-written (as explained in section 3.3.5). 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. We note that we are only able to perform this step after having evaluated that the generated DOM itself does faithfully capture the canvas state, and therefore the DOM distance comparison in this second stage of fault detection would faithfully capture incomplete or wrong canvas states.

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.

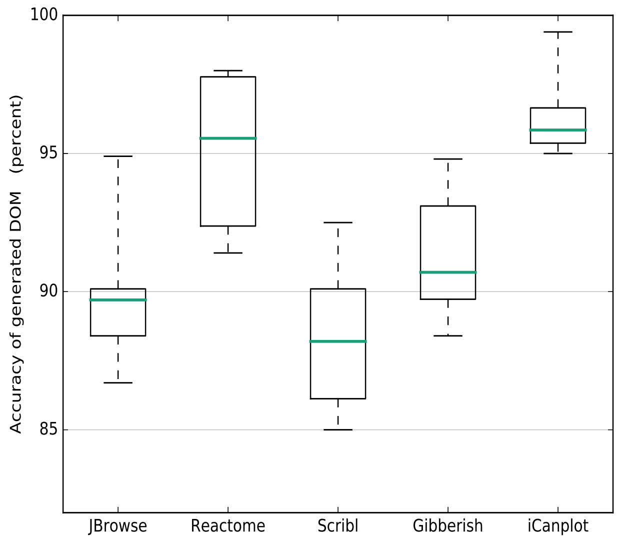


Figure 3.5: Accuracy of the inferred augmented DOM for each evaluated application in table 3.2. A total of 50 canvas elements in five different applications where used for this experiment.

3.4.3 Results and Discussion

Accuracy. Figure 3.5 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 3.5. 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 from 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 3.3) 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 3.3 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 3.4.2.

From Table 3.3, 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, some aspects in the fault injection process could be a source of internal threat to validity. There is currently no compilation of the types of faults that exist for canvases. Accordingly, in order to mitigate this threat and have an unbiased fault injection process, we chose to inject random shapes into the canvas. We also adopt 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. Furthermore, and perhaps most importantly, we do not have access to the objects on the canvas due to the lack of observable state in the canvas, which is the very problem that our approach is trying to solve. Accordingly, due to absence of access to canvas objects, simulating

Table 3.3: Results of detecting fault injections

Application	# Injections	# True Positives	# False Negatives
JBrowse [267]	20	18	2
Reactome [74]	20	20	0
Scribl [182]	20	16	4
Gibberish [214]	20	19	1
iCanplot [236]	20	20	0
Total:	100	93%	7%

a fault by removing a certain object is not practically achievable. 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 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 have no observable state 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 [29] 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.

3.5 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 [70, 147] 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,

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

WebDiff [63] 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 [170] 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 [149] 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 [52] 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.

3.6 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 chapter 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. We note, however, that the goal of this work is to provide developers with the fundamental capability of observing the canvas state and making assertions on it. However, it does not provide a complete testing solution, but rather make it *possible* to perform the testing process itself, thereby improving testability. As part of future work, a more through canvas testing solution can be provided such that it will cover more complex and resizable canvas elements, and generate the tests in a fashion that developers might prefer (e.g., using relative positioning in assertions).

Chapter 4

Semantic web accessibility testing

4.1 Introduction

Web accessibility is the notion of implementing web apps in a fashion that allows programmatic access to software functionalities that are otherwise only perceivable through certain senses (e.g., visually). Accessibility has been sometimes dealt with as an afterthought or a nice to have optional feature, with many developers and companies ignoring it altogether [114, 259]. However, as shown in Figure 4.1, millions of people around the globe have software-relevant disabilities and are impacted by web accessibility, or the lack thereof.

Furthermore, accessibility is increasingly becoming a legal requirement ratified into laws in many countries. For instance, in the United States, the Americans with Disabilities Act [253] requires all government and public agencies, as well as certain businesses, to make all their software and information technology services accessible. Similar provisions are required by law under the European Union's Web Accessibility Directive [204]. Figure 4.2 shows the increasing number of lawsuits filed in federal US courts against businesses for failing to provide accessibility accommodations.

Despite the increasing legal, economical, and human costs due to lack of accessibility, there has been little work in the software engineering research community to automate accessibility testing. Eler et al. [89] check for missing attribute fields or incorrect attribute values related to accessibility of web pages, an approach that is also used in a few patents [48, 226]. The bulk of existing work focuses on topics such as evaluating best practices for conducting empirical accessibility evaluations, such as manual checklist walkthroughs [46] or enlisting visually-impaired users as part of user studies [39]. A number of open-source tools have been developed to conduct simple syntactic accessibility tests that only check a few attribute values in a web page. In an audit of 13 of such accessibility testing tools conducted by the United Kingdom Government's Office of Digital Services [256], these tools found only 26% of a known small set of accessibility violations present in tested web pages.

The aforementioned tools are based on conducting *syntactic* checks, which are simple markup rules (e.g., any `a` element must contain a non-empty string) to check for a minimal level of accessibility. However, none of the aforementioned tools analyze key accessibility requirements related to the *semantics* of a page, such as the high level page structure and the roles or purpose of various elements. It is these semantic aspects of a page that users with disabilities rely on the most while using web pages [266], but none of the existing tools test for. Accordingly, the high level semantic analysis of web accessibility has remained a manual and laborious time consuming process [7, 8, 25, 47].

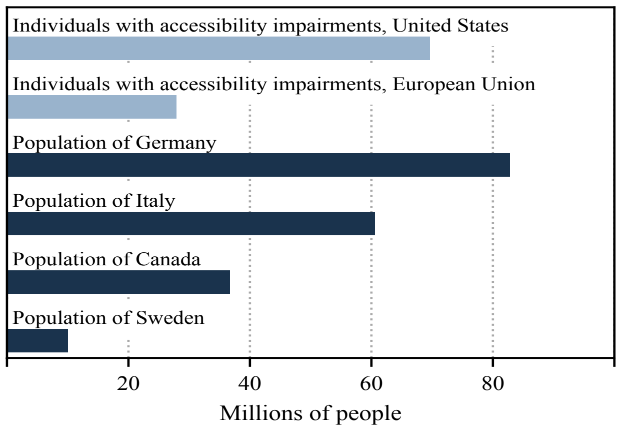


Figure 4.1: Number of people with software-related disabilities in the United States and the European Union. (Data compiled from: [59, 91])

To that end, we propose an approach that automates testing of a subset of web accessibility requirements pertaining to high level semantic checks that have not been amenable to automation. The approach is based on a visual analysis of the web page, coupled with a few natural language processing (NLP) steps, to conduct a semantic analysis of web pages. This analysis first identifies major cohesive regions of a page, then infers their *semantic role* or purpose within the page. Subsequently, a conformance check is conducted that determines whether the markup of the page correctly corresponds to the inferred semantics. If the markup contains the same semantic information perceivable by sighted users, the page is deemed accessible. Otherwise, if semantic information of the page is perceivable only visually but not conveyed in the markup, the page would be flagged as failing the accessibility test and the specific reasons are reported. In this work, we focus on vision disabilities as opposed to other forms of disability (e.g., hearing). The rationale for this is twofold. First, the web is predominantly a visual medium where

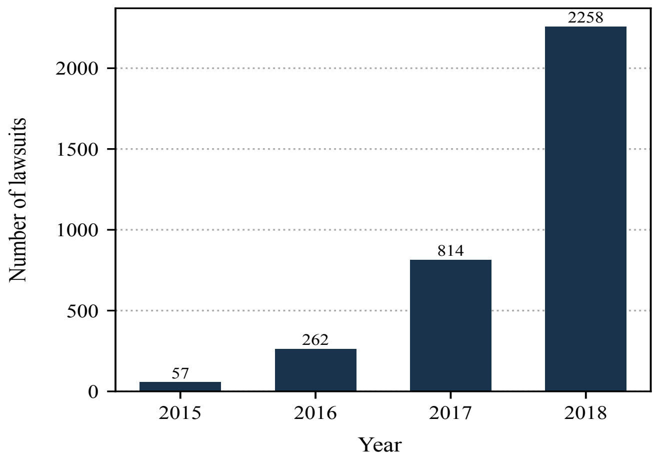


Figure 4.2: Number of software accessibility lawsuits filed in US federal courts per year. The number of lawsuits increased by around 3800% over the four-year period 2015-2018. (Data compiled from: [233, 257])

most of the information is accessed visually as opposed to other senses. Second, surveys have shown that vision disabilities are the most relevant to web users with disabilities [266].

This chapter makes the following main contributions:

- A novel approach for automatically testing semantic accessibility requirements, which is the first to address this issue, to the best of our knowledge.
- An implementation of our approach, available in a tool called AXERAY.
- A qualitative and quantitative evaluation of AXERAY in terms of its inference accuracy and ability to detect accessibility failures. The results show that it achieves an F-measure (harmonic average of precision and recall) of 87% for inferring semantic groupings, and is able to detect accessibility failures with 85% accuracy.

4.2 Background and Motivating Example

Figure 4.3 shows an example of an inaccessible web page. In a quick glance at the rendered page in Figure 4.3-(c), a sighted user can immediately understand the structure of the page and navigate their way through the various contents of the page. For instance, the user would immediately recognize that they can navigate to other areas of the web site through the navigation menu at the top (i.e., Home, News, FAQ).

While this happens naturally and instantaneously for sighted users, that is not the case for non-sighted users. The page structure (e.g., the presence of a navigation bar at the top) is *communicated exclusively* through visual design, since the HTML markup in Figure 4.3(a) is simply a collection of `<div>`s that do not communicate any semantic functionality. This implicit visual communication is intuitive and natural for sighted developers and users, but is unavailable for users who can not have *access* to visual information due to disabilities. Accordingly, the markup is deemed *inaccessible*, because it is expressed in a fashion that does not provide any semantic information about the page structure.

The analysis and conclusion that we just made is currently being done manually, since it requires high level semantic analysis of the page. Our goal in this work is to automate the reasoning we have just described in order to be able to automatically reach conclusions about the accessibility of web pages.

4.2.1 ARIA Roles

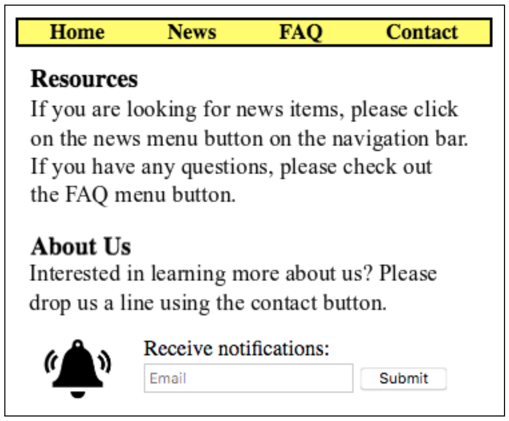
The aforementioned lack of semantic markup in the code makes it difficult for non-sighted users to navigate the page. This is because such users rely on *screen readers* to parse the page for them and present them with the various information

```html
<div class="nv8471">
<div>Home</div>
<div>News</div>
<div>FAQ</div>
<div>Contact</div>
</div>

<div class="_hd902">
Resources
</div>
<div>
 ...
</div>

<div class="_hd902">
About Us
</div>
<div>
 ...
</div>
```

(a) HTML markup

```css
.nav8471 {
   background-color:
    #ffff00;
   display:
    flex;
   justify-content:
    space-around;
   font-weight:
    bold;
   border:
    2px solid #000000;
}
._hd902 {
   font-size:
    5vw;
   font-weight:
    900;
   padding:
    8vh 0 2vh 0;
}
```

(b) CSS declaration



Home **News** **FAQ** **Contact**

Resources

If you are looking for news items, please click on the news menu button on the navigation bar. If you have any questions, please check out the FAQ menu button.

About Us

Interested in learning more about us? Please drop us a line using the contact button.

Receive notifications:

(c) Rendered page

Figure 4.3: An example of an inaccessible web page.

or navigation options present in the page. Screen readers are tools that speak out the various options, regions, or tasks accessible from the page, and the non-sighted user would then select one of the options they heard from the screen reader. Screen readers can be thought of as web browsers for non-sighted users, with one major caveat. While a standard web browser simply renders the page as is and leaves it to the end user to understand what the various elements mean, screen readers expect that the page markup contains semantic information about various areas of the page, which the reader would then announce audibly, giving non-sighted users an understanding of the overall semantic structure of the page.

The standard that is used by screen readers during their processing of web pages is the W3C *Accessible Rich Internet Applications* (ARIA) [1] semantic markup. The ARIA standard specifies a set of markup attributes that should be included in the page's HTML to make it accessible to screen readers or other assistive technologies. An example of such ARIA attributes is presented in the following subsection.

Targeted roles. ARIA defines more than 80 attributes spanning various aspects of the page, from low level syntax requirements to high level semantic structure. We therefore have to target a subset of these ARIA attributes because addressing all 80+ attributes in a single academic work is untractable. The basis for selecting the targeted roles is that we focus on the most commonly used subset of ARIA, which are *landmark roles* [266]. Our own experimental results (Section 4.4.2) also demonstrate the widespread use of these roles. The roles identify the major high-level regions of the page and specify the semantic role for each of them. They consist of a set of specific pre-defined roles that convey, though markup, the otherwise only visually perceived role of each region. For instance, the yellow rectangle at the top of Figure 6.1(c) is perceivable as being a navigation bar, therefore the markup of the element containing that region has to include the landmark attribute `role="navigation"`. When a non-sighted user loads the page, the screen reader would read out loud (i.e., generated audible speech) the presence of a navigation bar (and any other semantic regions in the page), allowing the user to quickly perceive the high level structure of the page, and directly interact with the region they are looking for.

Accordingly, the absence of semantic markup in Figure 6.1(a) makes the page inaccessible, because screen readers would be unable to provide alternative, non-visual, means of accessing the page. That is, the information and cues about the structure and navigation of the page remain locked in the visual design of the page and can not be accessed non-visually. When a web page communicates the important and key aspects of its structure and function using only visual design, without providing a programmatic means to access the same information, it is deemed inaccessible.

4.2.2 Existing Tools: Syntax Checkers

Existing accessibility testing tools [256] are based on syntactic checking. They check the HTML against certain syntax rules. For instance, one common check is to assert that every `img` element has an `alt` text attribute (for text descriptions). Another check asserts that `input` elements must not be descendent of `a`. Another example is checking that every `form` element has a `label` attribute.

While such syntax checks can be useful simple assertions and are easy to automate, none of the existing tools perform the more important, and more challenging, high level *semantic* analysis of the page's content. For instance, consider the sub-figure (c) in Figure 6.1, where we can see that there is a navigation bar at the top.

Now consider the HTML in subfigure (a), where we observe that the developer did not use the required ARIA attribute of `role="navigation"`. Accordingly, because we visually see a navigation region in (c) but do not find it expressed in the markup in (a), we conclude that the page has an accessibility failure, because we perceived a certain semantic structure as sighted users, but it was not expressed in the markup in order to ensure it would be equally available for non-sighted users. That is the the *central* issue in accessibility: making sure that whatever is visually perceived is also expressed in the code.

We now pause to reflect and observe that there is no syntactic check that can automate the conclusion we just reached. That is, the process that we just walked through requires our own visual perception as humans in order to reach the conclusion that there is a mismatch between the perceived page and the markup. This process cannot be expressed in the form of a syntax check.

In the next section, we describe how we construct an approach that automates the accessibility testing procedure that we have just manually conducted.

4.3 Approach

In this chapter, we propose an approach to automatically test for a subset of web accessibility violations that are pertinent to semantic structure. We recall that the scope of this work focuses on vision disabilities as opposed to other forms of disability, due to the web being a predominantly visual medium and the fact that vision disabilities are the most common software-related disability [266].

Figure 4.4 shows an overview of our proposed approach, which is based on the strategy of visually analyzing the web page to infer semantic groupings and their roles, and then checking that the HTML markup matches the inferred semantic

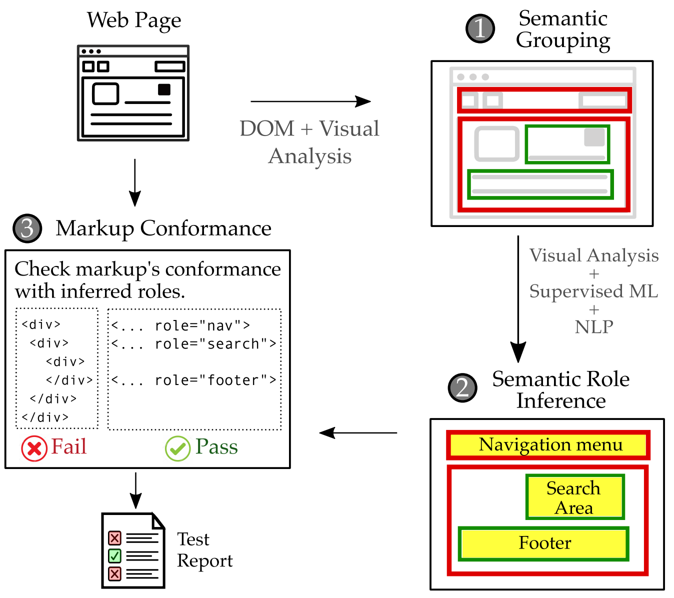


Figure 4.4: Overview of the proposed approach.

roles. The approach begins by obtaining the Document Object Model (DOM) and screenshot of the web page rendered in a web browser. Next, the set of visibly perceivable objects is identified. This is then used to perform a semantic grouping of the page into a set of semantically coherent regions. Subsequently, this information is used in an inference stage where the specific semantic role of each region is detected. Finally, the inferred semantics are checked against the markup used in the page, and a report is generated as to which parts of the page are inaccessible. The rationale behind this strategy is to check whether the semantics visually perceivable by sighted users are reflected in the semantics of the HTML markup, thereby ensuring accessibility.

4.3.1 Visual Objects Identification

In this first stage, the goal is to identify objects that are perceivable by sighted users, which we refer to as *visual objects*. For instance, in Figure 6.1(c), each item in the top navigation menu would be a visual object. This step of visual objects identification is the foundation of our overall approach, since the identification of visual objects enables checking whether the information and elements that are perceivable by sighted users are also accessible to non-sighted users.

Objects Extraction

We begin by taking as input the DOM of the page after it is loaded and rendered in a browser. We then extract from the DOM a set of nodes that represent visual content of the page, and we refer to each of these as *Visual Objects*. We define three types of Visual Objects: textual, image, and interactive.

Textual Objects. The extraction of text content is achieved by traversing text nodes of the DOM. More specifically:

$$\Theta_t := \{E \mid \nu(E) \wedge \tau(E)\} \tag{4.1}$$

where Θ_t is the set of all visual objects that represent text in the page, $E \in DOM$ is a leaf element iterator of the rendered DOM in the browser, $\nu(E)$ is a heuristic predicate that runs a series of checks to detect visually perceivable elements (as will be described in section 4.3.1), and $\tau(E)$ is a predicate that examines whether there is a text associated with E. More specifically, it returns non-empty nodes of DOM type `#TEXT`, which represent string literals. An example of extracted textual objects would be the "Resources" section in Figure 6.1(c). We note that the predicate is based on a node type, rather than an element (i.e., tag) type. This allows more robust abstraction because the predicate captures any text and does not make assumptions about how developers choose to place their text. In other words, regardless of the tag used for text data (e.g., `<span>, <div>`), text would still be stored in nodes of type `#TEXT`, even for custom HTML elements. This helps in making the approach more robust by reducing assumptions about tags and how they are used in the page.

Image Objects. Subsequently, we perform another extraction for image objects. We define this as follows:

$$\Theta_m := \{E \mid \nu(E) \wedge \mu(E)\} \tag{4.2}$$

where Θ_m is the set of all objects that are present in the page and represent images. As in the previous case, the predicate $\mu(E)$ examines whether there is any relevant image content associated with E. This has two possibilities: a) nodes of `<img>`, `<svg>`, and `<canvas>` elements, and b) non-image nodes with a non-null background image. An example of extracted image objects would be the bell icon in Figure 6.1(c). We note that this predicate makes the proposed approach more robust by eliminating assumptions about how developers markup images. If images are contained in standard tags (e.g., `<img>`, `<svg>`), then the predicate readily captures them. However, we make no assumptions that this is the only way an image can be included. For this reason, we also capture elements of any type when a non-null background image is detected.

Interaction Objects. Finally, we extract the interaction elements as follows:

$$\Theta_i := \{E \mid \nu(E) \wedge \eta(E)\} \tag{4.3}$$

where Θ_i is the set of all visual objects that represent form elements or similar interactive elements. These are determined by the predicate $\eta(E)$, which collects elements such as input fields and drop down menus. An example of extracted interaction objects would be the Email input field in Figure 6.1(c).

Visual Assertion

After the preceding extraction of an initial set of visual objects, this stage proceeds by conducting a visual analysis of the objects. This analysis detects if an object is visually perceivable. We conduct the visual analysis as follows. First, we obtain the box model of each object. We use the *computed* box model in order to faithfully capture the location as finally rendered on screen. Next, we obtain a screenshot of the region defined by the box model. We then analyze the screenshot using the Prewitt operator [194] used in computer vision. This operator applies a set of derivatives or differentiation operations on the image, and then typically used to detect salient visual features in the image (e.g., shapes, textures). We therefore use this operator to extract any visual features present in the image, regardless of the category or form of these features. Depending on the presence or absence of visual features in the image, the perceptibility state of the object is determined. If no visual features are detected, the object is deemed to be non-perceivable, and vice versa. For example, consider Figure 6.1(c). The navigation region in the top, and the main content that follows it, are perceivable by sighted users. However, web pages also have spacing elements that do affect the layout but are not individually perceivable themselves. For instance, there can be an element between the navigation bar and the "Resources" section such that a certain vertical distance is maintained below the navigation bar. While such a spacing element certainly affects the layout and occupies screen space, it does not constitute a visual object due to it's imperceptibility.

4.3.2 Semantic Grouping

After the visual objects identification is completed, we proceed by grouping visual objects into groups representing potential semantically relevant regions on the page. For instance, in Figure 6.1(c), one semantic grouping would be the navigation region at the top of the page. Another semantic grouping would be the "Resources" and "About Us" sections representing the main content of the page.

The rationale for this step of the approach is as follows. We recall that screen readers expect the markup to indicate the major semantic regions of a page. Accordingly, in order to automatically assert that any visually perceivable semantic region has been also expressed in the markup, we first need a mechanism by which we can detect the semantic regions in the first place. This is what we aim to achieve in this stage. Here we are only concerned with creating potential semantic groupings, while the next stage (Section 4.3.3) attempts to infer what is the semantic role (if any) of each potential grouping.

The grouping uses both structural (DOM) information as well as visual analysis. The DOM is used to generate a large number of potential seed groupings, and the visual analysis performs filtering and further analysis to produce a final set of groups. We adopted this strategy for the following reasons. We observed that the

DOM can potentially serve as a source of seed groupings. This is because of its inherently hierarchical nature that also tend to capture the developer's or designer's own intended semantic grouping. That is, the assumption here is that a set of nodes that has been grouped by a developer (i.e., implicitly via DOM hierarchy) is more *potentially* likely to represent some semantic value, compared to a random set of nodes. We emphasize that the groups only potentially have semantic value, and therefore serve only as an initial guess. The final decision of whether or not they do actually have a semantic role will be determined at a later stage (i.e., the role inference stage). Had we not performed this initial guess, the role inference alone would be practically untenable because of the extremely large combinatorial set of possible node combinations. Subsequently, visual analysis filters these initial seed groups and process them to construct semantic groupings. Visual analysis is used because, while the DOM may provide seed groupings, it does not faithfully represent what the end user is actually observing on the screen.

Grouping process. We now describe the mechanism of the grouping process. First, we obtain one flat non-hierarchical set of all DOM elements. For instance, in Figure 6.1(a), this would be all the `div` elements in one flat set. The elements are collected regardless of visibility, due to the complex nature of DOM and CSS rendering where non-visible nodes can contain visible children. For this same reason, the initial set of elements is flat and non-hierarchical, because visible children can often be inside non-visible nodes, and therefore relying on DOM hierarchy would yield many false positives and negatives. Instead, we build the hierarchy by visually analyzing the collected flat set of elements. We do this by first collecting the computed box model of each element in the set. For instance, in Figure 6.1(a), this would result in a set containing the computed box model of each `div` element regardless of hierarchy. Next, we remove box models that are visually located outside the page boundaries, since they are not perceivable to sighted users. For boxes that are only partially outside the page, we trim them to page boundaries. We note that we analyze the page as a whole, not only the currently visible portion. Subsequently, we filter *equivalent* boxes, which is when a pair of boxes visually contain the same set of visual objects. We do this by removing the smaller box in terms of visible area in a pair of equivalent boxes. This is because multiple boxes will often exist since many DOM elements can share similar visual dimensions and regions on screen. Next, we filter boxes based on how many visual objects are visually contained (i.e., located) within them. We remove each box that visually contains the entire set of visual objects on the page. For instance, in Figure 6.1(a), any `div` that visually contains the entire set of all `div`s is removed. This is because such a set does not represent any semantically useful grouping, since the entire set of objects is in one group only. Finally, we iterate over the set of visual objects. For each object, we find the largest box that visually contains the object. Once this is completed for all visual objects, the final result is a set of boxes representing the potential semantic groupings on the page.

4.3.3 Semantic Role Inference

Once semantic grouping is completed, we proceed to infer the semantic role of each group. This step infers one of the pre-defined landmark roles (Section 4.2.1). An example can be seen in the top navigation bar in Figure 6.1(c), indicating the pre-defined role of `navigation`. However, not all roles are relevant to our scope of automated semantic analysis. For instance, `region` is a generic catch-all label that does not convey any specific semantic role, and its use is generally discouraged and typically not used by screen readers. Another example is `form`, a label that indicates form regions. The label is directly associated with HTML `<form>` elements, and therefore no semantic analysis or inference is needed for its detection. Accordingly, we focus our semantic analysis on the more relevant roles of `main`, `navigation`, `contentinfo`, and `search`, which will be described in the following sections.

Main Role. The `main` ARIA role indicates a region that contains the main output or results in a web page. For example, on the search results page of a search engine, the region containing the list of retrieved search results would be the main region, which is then surrounded by other regions such as the navigation bar or footer.

The process by which we infer the role of a group to be `main` is as follows. First, we compute a score for each detected group in the page. The score uses both visual geometrical attributes as well as natural language processing (NLP) measurements. More specifically:

$$\psi_{main}(r) = A(r)\rho(r) \tag{4.4}$$

where r is a semantic grouping of the page, ψ_{main} is the score, $A(r)$ is the visual geometric area for r, and $\rho(r)$ is an NLP metric we define to measure linguistic aspects of the contents of r. More specifically, $\rho(r)$ first performs a part-of-speech (POS) tagging, which is a common NLP analysis than assigns POS labels (e.g., verb, noun, adjective) to each word. $\rho(r)$ then measures the variance of the linguistic POS tag frequencies of all textual objects contained in r. We give an example to clarify the various measured values. Consider the rendered page in Figure 6.1-c. r would represent, for instance, the region containing the body of the page (e.g., the Resources and About Us sections). $A(r)$ would be the geometric area of that region as visible on the screen. The rationale is to capture how much would a region occupy the visible space for sighted users. As for $\rho(r)$, it first collects all textual objects (as explained in section 4.3.1) within r, which would collect all text elements such as ”Resources”, ”About Us”, as well as the paragraphs on the page. For each text object, POS tags are collected, and then their frequencies (i.e., count of each tag type) are computed. ρ then measures the variance of these POS tag frequencies. For instance, a navigation region r that has, say, the textual objects “Images”, “News”, and “Settings” has no variance since they all have identical POS tags. Contrast this with the main body of text in a page, which contains elements such as such as paragraphs, section headings, links, and much more. The likelihood of all such content to be linguistically monotonous (i.e., all tags are nouns) is practically negligible. This is why eq. (4.4) includes the $\rho(r)$ factor. The $A(r)$ in the equation accounts for the fact that it is unlikely that the main region of the page would be the visually smallest area on the page. Once the score in eq. (4.4) is computed for all detected regions, we sort the regions by score and select the region with the highest score, which is finally reported to be the region having the main role. We note that we simply directly multiply the two factors, and do not have any thresholds or weights in order to have a parameter-free and more robust inference.

Navigation Role. The `navigation` ARIA role indicates a region in a webpage that allows users to navigate between various pages or views. The process by which we infer the role of a group to be `navigation` is as follows. We first compute a score for each group, using the following equation:

$$\psi_{nav}(r) = \frac{C(r)}{1 - \rho_h(r)} \tag{4.5}$$

where r is a semantic group of the page, ψ_{nav} is the score, and $C(r)$ is a metric that measures the *clickables ratio* inside the group r. This computes the ratio of visual objects that appear to be clickable to sighted users, which we define as any visual object whose onscreen cursor is a hand or a pointer, indicating to sighted users that it can be clicked on. Accordingly, a group that has high $C(r)$ is mostly composed of objects that a sighted user can click on, which is typically the case for navigation regions. For example, in Figure 6.1-c, the yellow navigation region at the top contains elements that all appear as clickables to sighted users. In contrast, a group that does not contain any clickables (e.g., only static texts and images) would have a $C(r)$ equal to zero and therefore is not a navigation region. This can be seen, for instance, in Figure 6.1-c in the body of the page below the navigation bar, where the body contains only static text paragraphs or images. $\rho_h(r)$ is a measure of the homogeneity of the contents of r. For semantic groups containing only textual elements, $\rho_h(r)$ is the same NLP linguistic variance metric we defined in eq. (4.4). For all other elements, $\rho_h(r)$ represents the dimensional variance of the objects in r. Finally, a given group is inferred to have a navigation role when ψ_{nav} greater than or equal unity, which was determined experimentally by manually testing this value empirically on a random group of websites.

ContentInfo/Footer Role. The `contentinfo` role (also known as the footer role) indicates regions of the page that represent complementary content to the parent document. That is, instead of containing the main output of the page or the main navigation elements, footer regions serve as complementary content or information that comes after the main content. In a similar fashion to previous roles, we compute a score for each detected grouping, using the following equation:

$$\psi_{footer}(r) = \frac{C(r)D(r)}{A(r)} \tag{4.6}$$

where, as in the previous roles, r is a semantic group of the page, ψ_{footer} is the score, $A(r)$ is the visual pixel count for r, $D(r)$ is the visual distance from the geometric center of r to the origin of the screen, and $C(r)$ is the clickables ratio in r as defined in eq. (4.5). As can be observed from the equation, the score is mostly concerned with the visual geometric aspects of the region, since this ARIA role is, by definition, spatial in nature since it refers to a specific spatial visual placement on the page. Accordingly, we compute and sort the score for all groups, and select the group with the highest score. If the group is located in the lower half of the page, it is reported as a footer. Otherwise no footer regions are reported.

Search Role. The `search` ARIA role indicates regions in a page that allow users to enter a search query and retrieve items on the page or site. To infer this role, we use a combination of visual analysis, a supervised machine learning model, as well as linguistic (i.e., keyword) techniques.

First, we train a Convolutional Neural Network (CNN) to visually recognize search icons. We collected and labeled 500 data points representing icon images (50% positive examples) and used the Inception CNN architecture [251], which has been shown to produce very effective classifications for computer vision machine learning problems [251]. Subsequently, we use this model to find search icons on a page. Next, we perform a nearest neighbor search to look for text input fields in the spatial vicinity of detected search icons. If a text input field is found, we mark the region containing the search icon and the input field as having a search semantic role. Furthermore, we also check for cases where the search input text field has no associated search icons. In this scenario, we extract all text input fields on the page. We then perform a nearest neighbor search to find any visible label texts in the visual spatial vicinity around the input field. We then conduct NLP *stemming* on the label text and find those that include key linguistically significant *stem words*, such as "find", "search", and "locate." Any detected group that matches any of the above cases is marked as having a search semantic role.

Finally, we note that, due to the non-hierarchical nature of our semantic groupings, all inferred roles are agnostic to hierarchies and the proposed approach is therefore able to detect hierarchical combinations of the inferred roles (e.g., a navigation region within a footer region).

4.3.4 Markup Conformance

This final stage asserts that the source of the page contains markup indicating the presence of the inferred semantic regions and their semantic roles. For instance, in Figure 6.1(c), the approach so far would infer that the group of elements at the top of the page represent a coherent semantic grouping, and that their semantic role is navigation. If the HTML markup corresponding to that area does not contain the ARIA landmark role of `navigation`, then screen readers will not be able to provide this semantic information to users, and we recall that this semantic information is among the most important and widely used of ARIA roles by users with disabilities [266]. Therefore, in such cases where the markup does not conform to the inferred semantic roles, we report an accessibility failure and indicate the expected semantic markup and where it should have been expressed in the page.

The mechanism of checking markup conformance is as follows. First, we obtain the semantic groupings and any inferred roles, as described in the previous sections. For each semantic group, we identify all DOM elements that satisfy two criteria: 1) all visual objects of the group are located inside the element's box model, and 2) the element's box model is located inside the group's box model. This process captures all possible DOM elements that would qualify as a root for the region, without including objects from other regions. Any of these DOM elements would therefore have to contain markup indicating the presence of a region and its role.

We then check whether any element in the set meets both of the following requirements: 1) the element has a `role` attribute whose value matches the inferred semantic role of the group. 2) the element's computed box model visually overlaps the box model of the inferred semantic group. The rationale for adopting this approach is as follows. As we noted in Section 4.3.2, the complex nature of DOM and CSS rendering easily allows cases where non-visible/non-rendered nodes can contain visible rendered children. A DOM-based approach (e.g., checking containment by XPath) would therefore yield many false positives and negatives. Accordingly, we use the visual check above for a more robust analysis.

If an element satisfying these requirements is found, we log it and move on to the next inferred semantic role and check that it has been correctly expressed in the markup. The process is repeated for all semantic groupings for which a role has been inferred. Any semantic grouping for which no role has been inferred is discarded. A report is finally generated indicating all roles that have been correctly expressed in markup, and all roles that should have been in the markup but are missing. We do not assume our approach understands the semantic roles better than what a human being would manually identify, and therefore there no miss-classification is reported if a markup already exists.

Implementation. We implemented the proposed approach in a tool called AXERAY (short for Accessibility Ray). It is implemented in Java. We use Selenium WebDriver to instrument browsers and extract DOM information and computed attributes. We use OpenCV [45] for computer vision computations, DeepLearning4J [88] for machine learning operations, and the Stanford CoreNLP library [172] for linguistic analysis. To make the study replicable, we made available online a link to our AXERAY tool and the anonymized participants' responses [183].

4.4 Evaluation

To evaluate AXERAY, we conducted qualitative and quantitative studies to answer the following research questions:

RQ1 How accurate is AXERAY in inferring semantic groupings and semantic roles?

RQ2 To what extent can AXERAY detect accessibility failures in real-world web pages?

In the following subsections, we discuss the details of the experiments that we designed to answer each research question, together with the results.

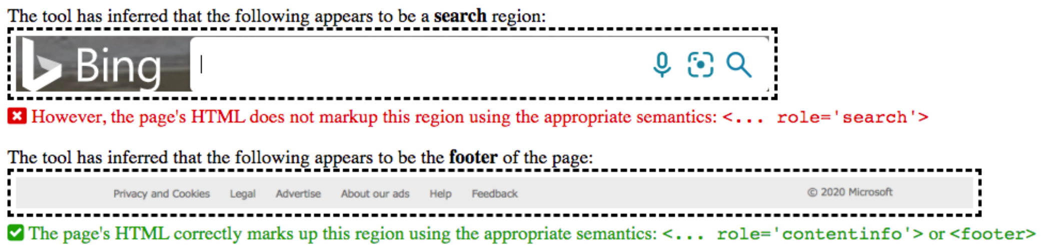
4.4.1 RQ1: Semantic Grouping and Roles Inference

In this question, the objective is to assess how accurate the semantic grouping and semantic role inference processes are. The rationale for evaluating this aspect is that the approach first performs the grouping and semantics inference, and then uses this inference to test for accessibility. Accordingly, we first need to assess the inference process itself.

We evaluated this question as follows. First, we collected 10 random subjects from the Moz Top 500 [1] most popular websites. Our approach does not have specific requirements for a subject other than being able to load it in a browser and access its DOM. We then ran the approach on each test subject's URL and obtained the output groupings and semantic roles. Figure 4.5 shows an example of the output. Each rectangle represents an inferred grouping, together with its semantic role. Subsequently, we recruited human evaluators. 10 evaluators were recruited from the MTurk [2] crowdsourcing platform. The qualifications of participants are to be working in the software industry and to have maintained the highest level of accuracy on the MTurk platform, which is referred to as Masters level.

Subsequently, each human evaluator was presented with the output of AXERAY for all test subjects, and asked to assess the accuracy of groupings and roles. More specifically, we asked them to identify any output groups that do not represent a meaningful semantic grouping. This represents the *false positives* of groupings, while the remainder are *true positives*. We also asked them to identify any meaningful semantic groupings on the page that were not included in the output. These are the *false negatives*. The same process is repeated for the semantic roles.

[1] https://moz.com/top500
[2] https://www.mturk.com


The tool has inferred that the following appears to be a **search** region:

✗ However, the page's HTML does not markup this region using the appropriate semantics: `<... role='search'>`

The tool has inferred that the following appears to be the **footer** of the page:

✓ The page's HTML correctly marks up this region using the appropriate semantics: `<... role='contentinfo'>` or `<footer>`

Figure 4.5: Sample of the generated accessibility report.

Results and Discussion

Table 4.1 shows the results of evaluating the accuracy of semantic grouping and role inference. The columns show the precision, recall, and F-1 measure (i.e., the harmonic mean of precision and recall) averaged across evaluators. The two groups of columns, labeled "Grouping inference" and "Role inference", show the accuracy of the proposed approach in inferring semantic groupings and semantic roles, respectively. The highlighted cells show the minimum and maximum values in each column.

The key outcome of this evaluation is the F-1 measures, which are at 87% and 90% for grouping and role inference, respectively. These values indicate a rather effective inference process. The lowest precision was 71%. This often happens due to a somewhat unusual DOM structure, where elements in the same region were placed at large tree depth separations from one another. This resulted in mistakenly grouping a number of elements that should have not been grouped. As for the recall, the lowest performance was at 60%. Such low values often happen in corner cases where elements are falsely excluded from groups due to having an empty box model stemming from complex nested CSS rules, despite being present in the group.

4.4.2 RQ2: Accessibility Failures Detection

While the previous question examined how accurate the inferred semantic groupings and roles are, this RQ evaluates to what extent the inferred information can be used to reliably detect accessibility failures.

Due to the absence of ground truth data that lists the actual failures per subject, we evaluate this RQ in two complementary ways: 1) using a fault injection experiment, and 2) evaluating the output on a large number of real-world subjects in the wild. Each evaluation is independent and separate from the other. The rationale for using these two complementary experiments is as follows. In the fault injection experiment (as will be explained in section 4.4.2), existing semantic markups (if any) are removed. This simulates an accessibility fault since, by definition, an accessibility fault is the *absence* of required markup. Subsequently, a check is made whether the tool is able to detect the absent markup and therefore detect the fault. The benefit of this experiment is that it allows automated evaluation without a subjective assessment. In other words, it creates a partial ground truth data from page developers' own tagging of roles. The drawback, however, is that it is only a *lower bound* of the actual accuracy, since it says nothing about other possible role faults that have not been injected (i.e., due to the absence of some roles from the original markup itself). For this reason, we supplement the fault injection experiment with a manual item-by-item evaluation of the inferred roles in order to evaluate all true/false positive and negative results. We describe each approach in the following subsections. We emphasize that the two experiments, fault injection and direct evaluation, are separate and independent of each other. That is, whereas the sub-

Table 4.1: Precision and recall of grouping and role inference. Highlighted numbers are the minimum and maximum values in each column.

	Grouping inference			Role inference		
Subject	**Prec.**	**Recall**	**F-1**	**Prec.**	**Recall**	**F-1**
wikipedia.org	94%	89%	91%	91%	91%	91%
google.com	88%	85%	87%	86%	81%	83%
amazon.com	92%	87%	89%	**83%**	**79%**	**81%**
stackoverflow.com	91%	94%	92%	91%	81%	86%
medium.com	96%	94%	**95%**	94%	**99%**	96%
khanacademy.org	92%	**98%**	95%	96%	89%	93%
imdb.com	**71%**	86%	78%	92%	96%	94%
cnn.com	**97%**	92%	95%	**97%**	97%	**97%**
rt.com	92%	**60%**	**72%**	94%	90%	92%
booking.com	92%	73%	81%	93%	89%	91%
Average	**90%**	**85%**	**87%**	**91%**	**89%**	**90%**

jects are injected with faults in the first experiment, all subjects are used in their original form for the second experiment.

Subjects. We conducted the experiments in this RQ on a total of 30 real-world subjects. The subjects were collected in two ways. The first half of subjects were randomly selected from the Moz Top 500 most popular websites as in RQ1. The second half of subjects were obtained from Discuvver.com, which is a service that returns a random website from the internet.

Fault injection

For the fault injection experiment, we inject a random fault in the markup of each subject, and assess if AXERAY was able to detect the fault. We recall from Section 4.2.1 that a web page is deemed inaccessible if there is an *absence* of semantic roles. That is, the developer did not add the necessary semantic markup to the page. Our goal is thus to simulate this behavior by *removing* all existing semantic markups on the page, and therefore create a new page as if the developer had

Table 4.2: Results of detecting fault injections.

Subject	Total # injections	# Detected faults proposed approach	# Detected faults baseline
bing.com	1	1	0
youtube.com	3	3	0
google.com	4	3	0
microsoft.com	7	5	0
bbc.com	5	3	0
amazon.com	4	3	0
medium.com	N/A		
yahoo.com	3	1	0
live.com	4	4	0
paypal.com	2	1	0
blogger.com	2	2	1
netflix.com	N/A		
stackoverflow.com	3	3	0
imdb.com	4	3	0
walmart.com	3	2	0
fuelly.com	1	1	0
typing.com	3	2	0
iconpacks.net	2	2	0
expatistan.com	N/A		
memrise.com	N/A		
retrevo.com	3	3	0
startupstash.com	4	3	0
eatthismuch.com	1	0	0
kdl.org	3	2	0
getpocket.com	2	2	0
retailmenot.com	3	3	0
mailinator.com	N/A		
myfridgefood.com	1	0	0
joinhoney.com	2	2	0
bannereasy.com	1	1	0
Total	71	**77.5%**	**1.4%**

not included the necessary semantic markup, and then check whether our approach can re-detect them. Such markup omission, by definition, is what makes web pages inaccessible, and is therefore the only meaningful fault type. In other words, any mutation of the markup is effectively a markup omission, since the exact expected semantic role would become absent from the markup. Due to the absence of ground truth information, the next best unbiased option we have is to assume that if a developer has taken the time to manually identify the role of each region, then we take that to be correct. We also note that misspelled attribute values are also effectively markup omissions. For instance, suppose that a region should have been marked as a navigation region. Whether the `navigation` role was completely absent from the markup, or was misspelled (i.e., `ngavoitn`), both cases are still effectively markup omissions.

We now describe the injection process. First, we load the subject in an instrumented browser (i.e., via Selenium). We then remove all semantic markups on the page (i.e., landmark `role`s). We then apply AXERAY on the subject and collect the output. If after the fault injection (i.e., removal of `role`s) AXERAY was able to indicate that there should be a semantic `role` (per each removed markup), we conclude that the injected fault has been caught. Otherwise, the fault was not detected.

Baseline. In order to have a more thorough evaluation, we included a baseline in our experiments. We could have just reported the results on their own, but adding a baseline offers some perspective on how relatively better the approach is. However, since there are no existing tools that perform semantic checking, our baseline consists of a simple random selection process. In this process, random regions from the page are selected. Next, a random semantic role is assigned to each randomly selected region. This set of semantic regions and their semantic roles is then taken to be the baseline.

Results and Discussion. Table 4.2 shows the results of evaluating the fault injection experiment. The first column shows the total number of fault injections performed on each subject. We recall that this first column is not a number we chose; it is rather the total number of injections that were *possible*, since the faults are removals of existing semantic markup. The second column shows the number of injected faults that were successfully detected by, whereas the third column shows the number of injected faults that the tool has failed to detect. The last row sums up the results across all subjects.

The main result of the evaluation is that AXERAY has detected, on average, around 77.5% of the injected faults. While this performance is relatively good, given that this sort of analysis hasn't been automated so far, we do note this is only a lower bound of the actual accessibility failure detection ability, since it says nothing about other possible failures that have not been injected (e.g., where the original markup itself did not include certain semantic roles).

In certain subjects in Table 4.2, marked with "N/A", the fault injection process was not possible. This was because the subject's markup did not contain any of the landmark semantic roles, and therefore it was not possible to remove them and check if our tool was able to detect them back. Despite some of these subjects being top 100 websites, the lack of such markup in the subject is an example that illustrates the need for effective and automated accessibility testing.

Direct output evaluation

In this step, we manually evaluate the output on a large number of real-world subjects in the wild. First, we loaded each test subject in an instrumented web browser (i.e., via Selenium). Page popups or notifications, if any, were closed. Next, we applied AXERAY on the subject and collected the generated output (as shown in Figure 4.5). We then categorized each item in the report into one of the following: *True positive:* This represents a true accessibility failure. This category holds whenever the tool has reported the absence of a correct semantic role that is indeed missing from markup, and therefore the reported failure is true. *False positive:* This is a false accessibility failure. In this case the tool has either reported an incorrect semantic role or the role is already in the markup but was falsely flagged as missing, and therefore the reported failure is false. *False negative:* This case is a false accessibility *pass* (not failure, as in the two previous cases). This represents cases where a semantic role should have been included the markup, as determined by manual visual examination of the page, but the approach did not report an accessibility failure. For instance, the page has a search region but the approach did not report that role. *True negative:* This corresponds to true accessibility pass. This represents cases where a role is actually semantically not present on the page, and the tool did not report a failure. This also corresponds to cases where a semantic role is present in the markup, and the tool has reported that the markup is conforming to the inferred semantic role.

Results and Discussion. Table 4.3 shows the results of the evaluation. Each row lists the subject, the true positive/negative and the false positive/negative for the subject. The last row shows the accuracy, precision, recall, and the F-1 measure.

Table 4.3: Direct evaluation of accessibility failure detection on 30 real-world subjects.

	Proposed approach				SortSite		Baseline			
Subject	**TP**	**FP**	**TN**	**FN**	**semantic issues**	**syntactic issues**	**TP**	**FP**	**TN**	**FN**
bing.com	4	0	1	0	0	11	0	3	0	4
youtube.com	0	0	4	0	0	10	0	2	0	0
google.com	2	2	3	0	0	6	0	2	0	2
microsoft.com	2	1	10	0	0	10	0	1	0	2
bbc.com	2	2	2	2	0	3	0	4	0	4
amazon.com	4	0	7	0	0	13	0	1	0	4
medium.com	4	0	1	0	0	6	0	2	0	4
yahoo.com	1	0	1	3	0	9	0	3	0	4
live.com	2	0	4	0	0	2	0	3	0	2
paypal.com	3	0	2	0	0	13	0	2	0	3
blogger.com	1	0	3	1	0	5	1	2	0	2
netflix.com	2	0	1	1	N/A	N/A	0	4	0	3
stackoverflow.com	3	0	6	0	0	18	0	2	0	3
imdb.com	2	2	3	0	N/A	N/A	0	3	0	2
walmart.com	6	0	2	1	0	7	0	4	0	7
fuelly.com	3	0	2	0	0	12	0	1	0	3
typing.com	2	1	3	2	0	10	0	2	0	4
iconpacks.net	3	0	1	1	0	8	0	4	0	4
expatistan.com	2	3	1	0	0	9	0	1	0	2
memrise.com	4	1	2	0	0	5	0	1	0	4
retrevo.com	2	0	3	0	N/A	N/A	0	2	0	2
startupstash.com	1	0	4	0	0	12	0	4	0	1
eatthismuch.com	6	0	1	0	0	15	0	3	0	6
kdl.org	3	1	2	1	0	9	0	3	0	4
getpocket.com	2	0	3	0	0	7	0	1	0	2
retailmenot.com	5	0	3	0	0	2	0	2	0	5
mailinator.com	2	1	1	0	0	6	0	4	0	2
myfridgefood.com	2	1	1	1	0	8	0	2	0	3
joinhoney.com	3	0	3	0	0	7	0	3	0	3
bannereasy.com	4	0	2	0	0	5	0	1	0	4
	Acc.	**Prec.**	**Rec.**	**F1**			**Acc.**	**Prec.**	**Rec.**	**F1**
	85.4%	84.5%	86.3%	85.4%			0.6%	1.3%	1.0%	1.2%

The values of the accuracy, precision, and recall are between 84% and 86%, indicating a relatively good performance. The true positive column represents the true accessibility failures that are indeed present in the subjects. This certainly does not cover all possible accessibility failures, but rather the subset of accessibility issues that we focus on in this work (i.e., the semantic roles). The true negative column can be thought of as the number of cases where the markup of the subject is "in agreement" with the tool's output. In the median, two true accessibility failures were detected per subject, and two inferred semantic roles per subject were in agreement with what has been expressed in the markup. For the second half of subjects in Table 4.3 (i.e., the random sites), 60% have used semantic roles. In contrast, for the subset of top websites (i.e., the first half of subjects), 74% have used semantic roles. 40% of the random websites did not use any semantic roles, compared to 26% of the top websites. This observation is expected since top sites are more likely to have more resources to create better products. The average execution runtime was 17 seconds.

We then investigated the reasons behind the false positives and negatives. One common reason is erroneous inference of navigation roles. This occurred, for instance, for a region that consisted of weather forecast for the next few days. From the perspective of our inference procedure, this looked like a navigation area since it had a group of links that were coherent in content and presentation (in the sense described in Section 4.3.3). Accordingly, it was falsely indicated as a navigation (i.e., a false positive), and therefore resulted in a false accessibility failure. Another reason involves missing footer roles that should have been reported. This occurred, for instance, for a region that was not recognized by the semantic grouping stage, and therefore no role was able to be inferred for it.

For comparison, Table 4.3 also includes an evaluation of SortSite[3], which is the best performing state-of-the-art accessibility testing tool [256]. As mentioned in the introduction, SortSite and other state-of-the-art tools only perform syntactic checks and is therefore unable to detect the semantic issues that are the focus of this work. Accordingly, we run an evaluation that verifies this empirically. Each subject is fed to SortSite, and the output report is saved. Each reported failure is then categorized as either a syntactic issue or a semantic issue. We recall that, as discussed in Section 4.2.2, a syntactic issue is any failure that only checks the syntax of the HTML. Examples include checks like ("each `a` element must contain non-empty text"), or ("`input` must not appear as a descendant of `a`"). These are direct checks that are only concerned with syntax. Compare this, for instance, with the reported failure shown in Figure 4.5. Here, the failure is *semantic* in nature. That is, the failure is not an application of some syntactic rule. Rather, the failure is reported because the markup does not conform to how the page *is semantically perceived* from a visual perspective, not because it didn't conform to a predetermined syntactic rule.

From Table 4.3, we observe that SortSite was able to find many syntactic issues (rows with N/A are cases were the tool was unable to load the subject). However, it did not detect any of the semantic issues. This is expected, as the rationale for this work was the observation that the state-of-the-art only conduct syntactic checks, which cannot detect the more important and widely used semantic information.

[3]https://www.powermapper.com/products/sortsite/

Future work

In this work, we focused on an important subset of accessibility requirements, which are the semantic roles as discussed in 4.2.1. Therefore, as expected, our approach can not cover all possible accessibility requirements. This leaves open a number of avenues for future work to address other accessibility requirements, each of which would require a novel technique to address. The variations in the semantics of various accessibility requirements and the lack of approaches to address them, mainly due to the difficulty of performing high-level semantic analysis, presents a rich and fertile ground to conduct research, which has received little attention from the software engineering research community as discussed in the introduction. For future work, we believe it will be a fruitful and interesting pursuit for the research community to explore some of these other accessibility requirements.

Threats to validity

We chose test subjects (i.e., web sites) randomly from the Internet with the mentioned criteria in Section 5.4, to avoid any selection bias. Plus, the participants were selected to be highly qualified evaluators at the crowdsourcing platform, mitigating the threats to the internal validity of the study. The subjects are diverse and complex enough to be representative of real-world scenarios, mitigating the external validity of the study by making the results generalizable. To make the study replicable, we made available online a link to our AXERAY tool and the anonymized participants' responses [183].

4.5 Related Work

Accessibility attribute checkers. Existing approaches related to accessibility testing focus on checking syntactical attributes. Eler et al. [89] and Patil et al. [201] check for missing or wrong UI attributes in Android apps, such as missing alternative text attributes in images, or color attribute values below a certain threshold. Similar checks are also used in other tools such as Google's Accessibility Scanner [107], WAVE [265], and ASLint [24]. The aforementioned works focus on syntactic checks, as opposed to the proposed approach in this work which checks for high-level aspects such as page structure and semantic landmarks, which are the most important ARIA roles that users with disabilities rely on [266].

Accessibility guidelines. The majority of existing work lies within the accessibility research community rather than software engineering. This research area involves studying certain categories of websites (e.g., airline websites [11, 85], education portals [135], other categories [42, 216, 232, 238]) or certain platforms (e.g., Android [18, 200, 276]), and then focusing on *manually observing* how non-sighted users would use those apps or websites in order to identify any patterns or trends in accessibility, with the purpose of publishing improved accessibility guidelines. Another line of work focuses on researching software development *best practices* and how do they impact the accessibility of the end product. For instance, Sanchez et al. [225] and Bai et al. [27, 28] examine development practices in agile teams working on accessible software, with the goal of proposing a guideline for better agile practices. Krainz et al. [141] investigates the impact of a model-driven approach to development on the accessibility of the created product. None of the aforementioned works, however, is concerned with developing an automated approach to test accessibility. Instead, their focus is researching best practices or guidelines for developers and designers.

Visual analysis. There exist a few techniques that analyze web applications from a visual perspective. Choudhary et al. [65] propose an approach that detects cross-browser compatibility by examining visual differences between the same app running in multiple browsers. Burg et al. [52] present a tool that helps developers understand the behavior of front-end apps. It allows developers to specify the element they are interested in, then tracks that element for any visual changes to understand code behavior. Bajammal et al. [32] propose an approach to generate reusable web components by analyzing design mockups. In contrast to our work, none of these works are related to accessibility.

4.6 Conclusion

Software accessibility is the notion of building software that is usable by users with disabilities. Traditionally, software accessibility has often been an afterthought or a nice to have optional feature. However, software accessibility is increasingly becoming a legal requirement that must be satisfied. While some tools exist to perform basic forms of accessibility checks, they focus on syntactic checks, as opposed to checking the more critical high level semantic accessibility features that users with disabilities rely on. In this chapter, we proposed an approach that automates web accessibility testing from a semantic perspective. It analyzes web pages using a combination of visual analysis, supervised machine learning, and natural language processing, and infers the semantic groupings present in the page and their semantic roles. It then asserts whether the page's markup matches the inferred semantics. We evaluated our approach on 30 real-world websites and assessed the accuracy of semantic inference as well as its ability to detect accessibility failures. The results show, on average, an F-measure of 87% for inferring semantic groupings, and an accessibility failures detection accuracy of 85%.

Chapter 5

Web form accessibility labeling

5.1 Introduction

Filling web forms is a common activity while browsing the web. The accessibility of web forms for users with disabilities requires the presence of markup that explicitly defines the form's structure and content. The markups, which are HTML attributes that are pre-defined by the World Wide Web Consortium (W3C) [1], provide a standardized and programmatic non-visual representation of the form, thereby enabling non-sighted users to access the form. In this chapter, we focus on markups of *web form labeling*, which is the explicit declaration of the labels for all fields present in a form. The absence of this form labeling markup is one of the most common web accessibility errors [123].

There is currently little to no proposed software analysis approaches that infer web form labeling. Having such a process would help in addressing a number of issues. First, the lack of web accessibility impacts millions of people around the world [59, 91]. Such users would benefit from a software analysis approach that automatically makes web forms more accessible, especially because missing labeling markup is one of the most common accessibility errors [123]. For developers, it has been shown [114, 259] that the general adoption of accessibility in some software projects has been low due to other competing and pressing development and business requirements, and therefore having an analysis approach that can fix form labeling errors would be helpful in automating and reducing workload. Furthermore, accessibility is increasingly becoming a legal requirement for software products [233, 257], and therefore research towards automating some of the accessibility analysis would reduce the workload on developers and their companies. Accordingly, for the aforementioned reasons, proposing an automated form label analysis approach would be an important addition to the software analysis literature.

Existing approaches perform testing of accessibility markups, but do not conduct labeling inferences to repair the form and make it accessible. Eler et al. [89] check for missing attribute fields or incorrect attribute values related to accessibility of web pages in general, an approach that is also used in a few patents [48, 226]. A number of tools have also been developed to check for and improve various subsets of markup [22, 280], but do not infer or repair missing DOM labeling markups. The bulk of existing literature focuses on non-software engineering topics such as evaluating best experimental practices for conducting empirical accessibility evaluations [18, 39, 42, 46, 238] None of these works propose any software analysis techniques for form labeling. Accordingly, the existing standard practice for repairing software accessibility errors has remained a manual laborious process [7, 8, 25, 47, 280].

In this chapter, we propose a web form analysis approach that automates the repair of form labeling markup. The approach is based on visually analyzing the content and structure of a web form, and then augmenting the inferred labels into the Document Object Model (DOM). This analysis first converts the form into a set of abstract elements in order to streamline subsequent analysis. Next, a set of cues consisting of visual prominence and layout cues are constructed from these elements. Subsequently, the labeling process is achieved by solving a set of objective functions and constraints in order to reach the final labeling associations. Finally, these associations are converted into standard labeling markup in order to make the form labels accessible.

This chapter makes the following main contributions:

- A novel web form analysis approach for automatically inferring form labeling markup, which is the first to address this issue, to the best of our knowledge.

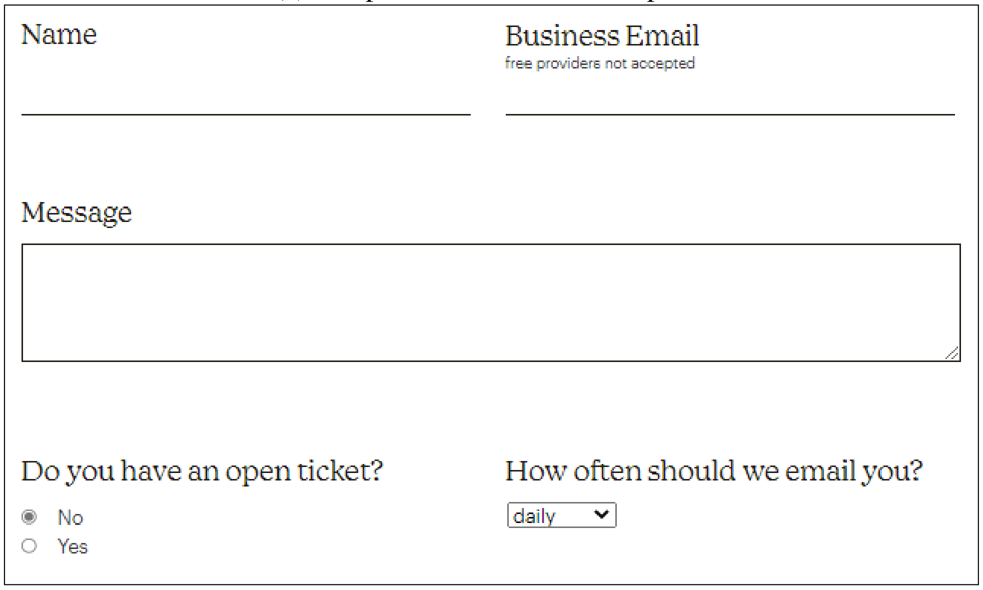
- A technique that is based on a combination of visual cues and optimization formulation to repair the DOM of a web form in order to have accessible labels.

- A qualitative and quantitative evaluation in terms of accuracy, safety, and performance. The results show an average F-measure of 88.4% for labeling accuracy, and an average run-time of around 1.6 seconds.

- An implementation of our approach, available in a tool called AXEFORM [183].

```html
<form>
<section>
 <div><p>Name</p></div>
 <div><p>Business Email</p><span>free providers not accepted
     </span></div>
</section>
<input type="text" id="vr_9481">
<textarea id="bx_3978"></textarea>
<input type="text" id="vr_6588">
<div><p>Message</p></div>
<section>
 <div><p>Do you have an open ticket?</p></div>
 <div><p>How often should we email you?</p></div>
 <span>Yes</span><span>No</span>
</section>
<input type="radio" value="Y" id="op_y">
<input type="radio" value="N" id="op_n" checked>
<select id="frq"><option>daily</option>
 <option>weekly</option><option>monthly</option>
</select>
</form>
```

(a) Sample of the HTML markup



Name

Business Email
free providers not accepted

Message

Do you have an open ticket?
● No
○ Yes

How often should we email you?
daily

(b) Rendered form

Figure 5.1: An example of an inaccessible web form.

5.2 Background and Motivating Example

Figure 6.1 shows an example of an inaccessible web form. By looking at the rendered form in Figure 6.1b, a sighted user can understand the structure and content of the web form and navigate their way through it. For instance, the user would recognize that they can write down their name, email, and indicate if they would like to receive daily emails. Most importantly, the user recognizes which form fields are associated with the name or the email, for instance. The collection of information we just described is referred to as form labeling. It indicates what fields are present on the form and specifies the textual label that describes what each field represents.

While this understanding of form content and labeling happens naturally when sighted users look at the form, that is not the case for visually impaired (i.e., non-sighted) users due to the absence of visual perception. In *inaccessible* forms, such as the one in Figure 6.1, the labeling is communicated exclusively through visual design. This is because the HTML markup in Figure 6.1a is just a collection of `<div>`s, `<p>`s, and `<input>`s that do not communicate any labeling associations to indicate what the various elements represent. For instance, in Figure 6.1a, there is no piece of markup indicating what the `<textarea>` represents or means. That is, there is no markup that describes to a non-sighted user what information they are supposed to provide for that form field. The form is therefore unusable in this case. In contrast, sighted users understand that the `<textarea>` is where they should write down their message, or that the first text input is where they should provide their name. They understood this from the visual design and layout of the form. This implicit visual communication is natural for sighted developers and users, but is unavailable for users who can not have access to visual information due to disabilities. Accordingly, the web form is deemed inaccessible since its markup does not specify the labels of the form elements.

5.3 Approach

The objective of this chapter is to propose a web form analysis approach to automatically create the required DOM markup for accessibility labels. The challenges involved in formulating such an approach are as follows. First, web pages and forms can have many possible designs and patterns. Designing an approach that addresses every potential case is not tractable. Therefore, the goal is to first establish an approach that works for many forms while remaining generic and robust. This is an important first step since the problem of inferring labeling markup has not been addressed before, so the first stage is to establish a technique that can be demonstrated to work for many forms, after which other corner cases can be addressed by permutations and adjustments in subsequent publications. The second challenge is the absence of prior blueprint or similar work that can serve as a starting point to formulate the approach. It is therefore not readily evident what assumptions are safe to make or what sort of analysis steps or general analytic approaches should be utilized to address the problem. Accordingly, the approach will be mainly heuristic in nature, and its efficacy will ultimately be assessed in the evaluation.

The overall steps of our approach are as follows. We adopt a strategy of visually analyzing the web form to infer labeling associations of form elements, and then repairing the DOM such that its markup reflects the inferred labels in order to make the form accessible. The approach begins by loading the web page in an instrumented web browser (e.g., via Selenium). Next, any forms on the page are then abstracted in preparation for further analysis. Subsequently, a set of visual cues are constructed from each form. These visual cues are then used in the next stage, which is the optimization formulation. In this stage, we cast the labeling process as an optimization problem, where the minimization and maximization of certain combinations of visual cues yield the final form labels. Finally, the inferred labels are transformed into standard ARIA accessibility attributes and augmented into the DOM of the web page.

5.3.1 Form Abstraction

The abstraction process extracts essential information from the form by converting the form into only two classes of objects: *fields* and *texts*. Any other aspect of the form is discarded.

For fields, we iterate through the possible ways in which form inputs can be expressed (e.g., `input` elements with various attributes, `select` elements) and abstract them into a single category of fields. When it is time to perform the optimization and generate the labels, as described in Section 5.3.2, working with a single category of fields facilitates a more manageable optimization formulation instead of working with a variety of categories.

For texts, we also perform a similar abstraction. Labels in web forms can be expressed using many possible textual elements. For example, in Figure 6.1, `p`, `div`, and `span` have been used. We normalize these variations and abstract the texts by iterating through the DOM tree and selecting non-empty nodes of type `#text`, which represent string literals. We note that this is based on a node type, rather than an element (i.e., tag) type. This allows more robust abstraction because it captures any text and does not make assumptions about how developers choose to place their text. Regardless of the tag used (e.g., `span, div`, attributes), text would still be stored in nodes of type `#text`, even for custom HTML elements.

5.3.2 Labeling Association

The labeling association is achieved by gathering visual cues from the abstracted form, then using these cues to construct the decision variables that will make up the objective functions as well as the constraints.

Visual Cues

When sighted users browse a web form, they are able to perceive the form labels and understand what each field signifies. Our goal is to emulate this perception and analysis in an automated fashion. However, due to the absence of prior analysis approaches for inferring form labels, we do not have a reference or general blueprint for what sort of analysis steps may be performed to infer the labels or what assumptions are safe to make. However, an assumption that is guaranteed to be accurate is that regular users take visual information as an input, and build their mental understanding of the form as an output. Accordingly, we use this assumption as the basis for analysis. That is, we propose a web form analysis approach that emulates users by collecting visual information from the form as an input and generate the labeling association DOM markup as an output. The next question then becomes what exactly is the kind of visual information that should be collected.

Unfortunately, there is no conclusive answer to this question yet. It is not definitively known what kind of visual information sighted people intuitively use when deciding the labeling in web forms. Furthermore, as mentioned in the introduction, web forms can vary greatly in their designs and patterns, which makes it difficult to propose a solution that can cover all possible form layouts. However, two particular types of visual cues may potentially be useful when analyzing web forms to infer label markup, as will be described in the next paragraph. While these visual cues do not cover all possible form designs, they do serve as a first step towards solving the problem of accessibility labeling inference. We first emphasize an important aspect of the proposed analysis, which is that no single cue is sufficient or conclusive on its own. Rather, it is the collective optimization of all possible cue combinations that will yield the final DOM labeling markup. This combination of cues mitigates the fragility of depending on a single cue on its own, as will be discussed in section 5.3.2.

The first cue is related to how much an object stands out for a user who is looking at the interface. We will refer to this type of information as *visual prominence*. It has been shown that users looking at a web page do not pay equal attention to all objects on the page [90, 235]. Rather, objects with certain visual characteristics are more likely to gain their attention compared to other objects. It is therefore likely the case that if we gather the kind of visual characteristics that users intuitively focus on, then we would be capturing potentially useful cues for analyzing a web form. To this end, we capture the following visual characteristics to analyze the prominence of various form elements. First, users have been shown to focus on objects that are bigger and occupy more space on the screen [53]. Accordingly, by capturing size information of form objects, we will be incorporating in our model a feature that users tend to focus on when parsing the form. Subsequently, existing evidence [108] suggests that objects with higher contrast tend to receive more focus from users. We therefore add this visual cue as well to the list of data to collect from the form. Finally, we capture all the aforementioned visual cues into a single metric of visual prominence. This metric is constructed by the multiplication of three measurements: font size, weight, and contrast. First, each of these values are captured for every abstracted form element (as described in Section 5.3.1). We then take the maximums of each of these values and use that to compute a normalization factor. Finally, all visual prominence values are divided by this normalization factor to yield the final normalized visual prominence values. Only these normalized values are used in the remainder of the analysis. More concretely, the visual prominence is computed as follows:

$$p_i = \left(\frac{s_i}{n_s}\right)\left(\frac{w_i}{n_w}\right)\left(\frac{c_i}{n_c}\right) \tag{5.1}$$

where p_i is the visual prominence of the i-th abstracted form element, s_i and w_i are the font size and weight, c_i is the color contrast, and n_s, n_w, n_c are the normalization factors computed from taking the maximum value across all abstracted form elements. Figure 5.2(a) illustrates the nature of the visual prominence cue, where the thickness of the line indicates how prominent the various elements are (i.e., it visualizes the p_i described above). We also note how we do not assume some factors are more important than others, and therefore make the unbiased decision do weight all factors equally. We can see how the “free providers...” text receives a lower prominence value compared to the “Business email” text.

The second cue is related to the visual *layout* of elements on the form. This would provide information on how elements are arranged with respect to one another to capture the layout of the form. The reason we believe this information would be useful for labels inference is because there is evidence [37, 270] that users tend to associate form inputs with their neighboring text fields. Accordingly, this information may potentially facilitate emulating the kind of visual parsing done by sighted users. However, we do emphasize that web forms can vary greatly in their designs and layouts, and therefore it is very difficult to propose an inference process that covers all possible form layouts. This is the reason that we don't only use the layout cue on its own, since it would not necessarily be conclusive on its own. Rather, we combine multiple cues in order to reach final labeling decisions. We capture this cue by measuring the pair-wise visual geometric Euclidean distance, which is used together with other cues as explained in the previous paragraph. This is done across all abstracted field-text pairs. After each pair-wise measurement is collected, we proceed to determine the geometric bounds of all measurements. The reason for doing this is to normalize the geometric measurements. The reason for doing the normalization is because this cue will be combined with other cues in the objective as well as the constraints of the optimization. Accordingly, for the optimization to function properly, the geometric measurements must first be normalized, because other cues (e.g., prominence) would have a different range of values compared to layout distance, and therefore combining them directly before normalization would skew the optimization towards one cue instead of another. To do this normalization we compute the geometric bound, which is the maximum possible pair-wise Euclidean distance for a collection of elements, and is therefore calculated once all distances are available. Finally, all distances are divided by the geometric bound and the resulting values are the final normalized values used in the remainder of the analysis. More concretely, the visual layout of the form is a set of data whose elements can be expressed as follows:

$$l_{i,j} = \frac{E(f_i, t_j)}{n_g} \tag{5.2}$$

where $l_{i,j}$ is the pair-wise visual layout cue, E is the visual geometric Euclidean distance between field f_i and text t_i, n_g is the normalization factor computed from the geometric bound as explained in the previous paragraph. Figure 5.2(a) illustrates the nature of the visual layout cue, where the length of the line visualizes the $l_{i,j}$ values described above. The collection of all such pair-wise lines captures the layout of the whole form.

Decision Variables

Decision variables will be used, in combination with the cues gathered from the form, to optimize the assignment of labels. The strategy we adopt is to formulate the form labeling analysis as a constrained binary program. The rationale for formulating the problem in this fashion is as follows. First, the assignment of labels must be constrained by other label assignments. That is, if a solver has made labeling assignments involving some subset of texts, then this should constrain the possible set of texts that can be used for remaining labeling assignments for the rest of the fields. Furthermore, the labeling assignment can be readily expressed as a pair-wise binary function between fields and texts. In addition, deciding the form labeling using only individual field by field analysis would be a subpar approach. This is because it would not consider whether a potential label for a given field might be a better match for another field, or whether it was already associated with another field. For instance, if field A has equal cue values with respect to

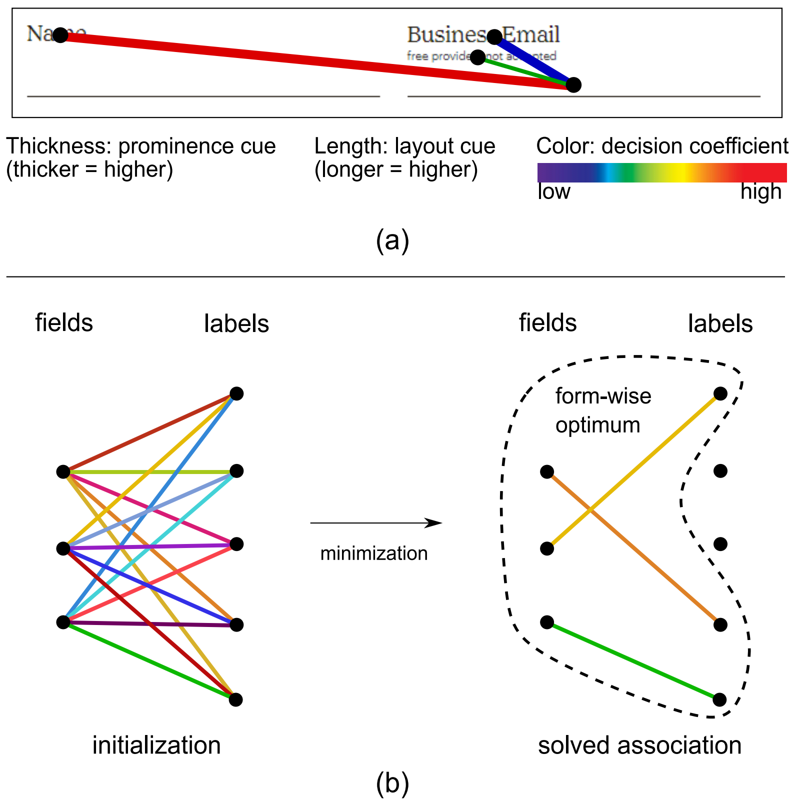


Figure 5.2: Visualization of (a) the visual cues, and (b) the labeling association.

potential labels X and Y, but field B has optimum cue values only with respect to Y, then the optimal decision is to assign A to X and B to Y. Having an isolated, field by field analysis would not be able to make this decision. In contrast, a global combinatorial analysis that takes into account all fields together would be a better fit. This is also the reason why the proposed approach does not rely on individual cues alone to decide field labeling, but rather trades off all cues for all field-label pairs to reach an optimum labeling for the form as a whole. For all of the aforementioned aspects, a constrained binary program is a potentially suitable approach to formulate the labeling problem.

Accordingly, each decision variable $d_{i,j}$ is a binary value that represents the decision of assigning a particular label t_j to a particular field f_i. Figure 5.2(b) illustrates the decision variables. On the left hand side of the figure, each line represents one $d_{i,j}$ value. Each variable would encode one possibility of labeling. The optimization solution, as illustrated on the right hand side of Figure 5.2(b), represents which decision variables are finally applicable. This is where the tradeoff occurs in terms of optimizing various possible combinations of visual cues to reach the final labeling. Once all decision variables have been determined, the label associated with each field is identified.

Objective and Constraints

In order to create the objective function and constraints, we need to consider the kind of labeling markup provided by the ARIA standard. This is because the final DOM repair process will include generating the necessary markup to convey the inferred labels, and this markup has to follow the ARIA standard. The ARIA standard provides two types of form labeling: individual and group labels. Individual labeling represents the label associated with every single form input on its own. For instance, in Figure 6.1b, individual labeling associates the large text area field under 'Message' with the label ‘Message‘. For group labeling, consider the two radio option fields at the bottom of Figure 6.1b. The individual labels in this case are 'No' and 'Yes', and the group label is ‘Do you have an open ticket‘. The strategy we adopt to identify both label types is a bottom-up multiple-passes approach. We first infer the individual labels, and then use that information to perform another pass to identify the group labels. The reason for adopting this strategy is that it allows us to optimize the problem for each label type separately, and also simplifies group labeling by using information from the individual labeling. Therefore, each of these label types will have a different objective function and set of constraints.

Subsequently, each decision coefficient of the optimization is calculated as follows:

$$c_{i,j} = \frac{l_{i,j}}{p_i} \tag{5.3}$$

where $c_{i,j}$ is the coefficient of the decision variable, p_i is the visual prominence cue, and $l_{i,j}$ is the visual layout cue. The magnitude of each coefficient $c_{i,j}$ describes the pair-wise and per-element visual cues. The choice of which i, j pairs would constitute a labeling association is determined by the objective function and the constraints:

$$\begin{aligned} A \;=\; & \text{argmin} \sum_i \sum_j c_{i,j} d_{i,j} \\ \text{s.t.} \quad & \sum_j d_{f_1,j} = 1, \cdots \sum_j d_{f_N,j} = 1 \\ & \sum_i d_{i,t_1} \leq 1, \cdots \sum_i d_{i,t_M} \leq 1 \end{aligned} \tag{5.4}$$

with A being the labeling associations result. c and d are the coefficients and decision variables, respectively, as described in the preceding paragraphs. Figure 5.2(b) illustrates a number of remarks regarding the above equations. First, the color of the lines represent the magnitude of the decision coefficients $c_{i,j}$. We can see that the visual cues cover all possible labeling combinations. The decision variables then determine which combination of visual cues yields the optimal result, as shown on the right hand side of Figure 5.2(b). We also note that we have two sets of constraints. The first set of constraints subjects the optimization to the requirement of finding exactly one label. This constraint stems from the ARIA accessibility requirement stating that each form field must have a label, and is therefore not assumption or decision that we made on our own. In Figure 5.2(b), only one outgoing line is allowed from each field dot. Each form field f gets its own constraint equation, containing entries for all possible labels. First, this prevents the optimization from yielding the trivial solution (i.e., none of the lines in Figure 5.2(b) are selected). Second, this avoids solutions where only a few fields are associated with all labels by virtue of yielding a smaller objective value, thereby leaving many other fields without a labeling association. For instance, in the solved association in Figure 5.2(b) we note that the final selection coefficient values are *not* the minimum per each field. That is, the final lines on the right hand side are not all blue colored that indicate low coefficient values. Rather the final coefficient values are mid-range, illustrating that the objective function is seeking the minimum of the whole form, not the individual fields. The second set of constraints subjects the optimization to restrictions on the possible associations that can be done for text elements. Each text t gets its own constraint equation, paired with all possible fields. Paired with the first set of constraints, these second constraints enforce a couple of requirements. First, that the labeling association for each text entry is optional. That is, it encodes the expectation that not every text field present on a form has to be a label. We can see this in the solved association in Figure 5.2(b) where not all the label dots have lines. This is true because forms have many texts that are not labels (e.g., descriptions, hints). Furthermore, the constraints ensure that each potential label is associated with at most one form field. This enforces the requirement that a solution is not accepted if only a few text elements end up being associated with many fields due to a smaller objective value.

Once the individual labeling associations are obtained, we proceed to infer the group labeling. First, we discard all texts that have received a labeling association from the individual labeling stage (i.e., all $t \in A$). Next, our target is to identify groupings of the fields. To achieve this, we take an agglomerative clustering approach [99], which is a clustering technique that builds clusters by starting from leaf nodes and than gradually merging groups of nodes. The reason for adopting agglomerative clustering is that we use it in a way that enables us to identify groupings without requiring thresholds or parameters. This helps in increasing the robustness of analysis, and reduces the often cumbersome manual effort required to tune any parameters.

We conduct the grouping by first constructing a fields distance matrix, whose elements are the visual geometric Euclidean distance between each pair of fields. Subsequently, we determine the linkage function to be used for the agglomerative clustering. This function determines which pair of clusters to merge in the next level of agglomeration. For the current task of identifying visual groupings, we adopt the single-linkage criterion [99]. Single-linkage agglomeration occurs, in our case, when the minimum visual geometric distance between clusters is smallest.

Accordingly, we run the agglomeration process as per the aforementioned steps. We then obtain all heights of the obtained hierarchy of clusters. In our case, these heights would signify the progressively increasing visual geometric distances between clusters. We then compute the median value of all heights, excluding heights of zero since they don't represent any cluster. This median value is then used as a cutoff to the hierarchy of clusters. That is, only clusters whose within-cluster visual geometric distance are below the cutoff are retained. The final result is a set of field groupings. Once the groupings have been identified, we proceed to infer the labeling associations for those groups. We achieve this by performing a second optimization pass and formulate the problem as follows:

$$\begin{aligned} B \;=\; \text{argmax}\; & \sum_i \sum_j \frac{d_{i,j}}{c_{i,j}} \\ \text{s.t.}\quad & \sum_j d_{f_1,j} \leq 1, \cdots \sum_j d_{f_N,j} \leq 1 \\ & \sum_i d_{i,t_1} \leq 1, \cdots \sum_i d_{i,t_M} \leq 1 \end{aligned} \tag{5.5}$$

where B is the labeling associations result, and c and d are the same variables as those that have been used in the first optimization pass, with the exclusion of all individual labeling associations that have been obtained from that first pass. We make a number of remarks regarding the above equations. First, we note that the structure of the problem is different from that in the first optimization pass. We now have a maximization problem instead of a minimization. The rationale for this is as follows. First, we note that individual labels are always present while group labels may or may not be present [1]. The problem formulation therefore needs to take this into account. This brings us to the second observation, which is regarding the constraints. In the previous pass, we had constraints to have the decision variables finding exactly one label (i.e. the equality constraints). In this pass, however, no such requirement is made. Instead, we have an upper bound inequality constraint on the decision variables pertaining to all form fields (the first set of constraints in Equation (5.5)). This encodes the fact that group labeling associations may or may not be present on a given form, and if it does have one, that its doesn't result in a degenerate case where many texts are associated with a given field. However, since we now have an upper bound inequality instead of an equality constraint, the problem can no longer remain a minimization. This is required in order to avoid trivial solutions (i.e. zero decision variable selections). Accordingly, the problem is formulated as a maximization. The placement of the coefficients of decision variables have been adjusted accordingly to reflect the change in the objective. However, their computation remains the same as before, using the same visual cues we previously described. We also note that the second set of constraints (pertaining to the text elements) have remained the same as before, since we still have the same expectation as the first case that not all texts need to partake in a labeling association.

5.3.3 DOM Augmentation Markup

So far, the form has been abstracted, visual cues generated, and the optimizations yielding the labeling associations have been solved. At this stage, we perform the final step needed to ensure form labeling accessibility, which is repairing the DOM. This stage involves transforming the labeling associations into ARIA accessibility markups. These markups are finally added to the DOM of the page in order to explicitly encode all labels.

The ARIA standard provides markup for both individual and group labels. We will start by describing the encoding for individual labels, and then proceed to describe to the case of groups. First, we take all the individual labeling associations obtained in A (Equation (5.4)). This provides a mapping of f_i to t_i, representing pairs of fields and labels, respectively. Next, we iterate over each field f_i, and examine the associated label t_i. If the t_i element in the page has an id (i.e. a non-zero `id` attribute), then we obtain that id. Otherwise, a unique random id is generated for that element and added as an `id` attribute for that element in the DOM. Subsequently, for each field f_i, we add the `aria-labelledby` standard ARIA attribute. The value of this attribute is then set to the value of the `id` attribute of the associated t_i, whether that id already exists or was generated. If the associated t_i is not an explicit element (e.g., the text is contained in an attribute), then we add an `aria-label` instead of `aria-labelledby`, and we set its value to be the content of the text, since assigning an id in this case is not possible. The ARIA standard allows labeling elements using both ids and the actual textual value of the label.

We then move on to process the group labeling associations obtained from B (Equation (5.5)). In this case, the mapping result is different from the case of individual labeling associations. Here the mapping is from a set of fields $\{f_1, \ldots, f_n\}$ to the associated inferred label t_i. We begin by iterating over each set of fields, and examine the associated label. We obtain the id of the associated label, or generate a unique one if it does not exist, in the same fashion as done in the previous case for individual labels. Next, we find the *nearest common ancestor* in the DOM for the $\{f_1, \ldots, f_n\}$ set. The reason for doing this is the nature of the standard ARIA attribute used to group related elements, which is `role="group"`. It should contain as descendants the group of elements its representing [69]. Accordingly, for the nearest common ancestor node of a set $\{f_1, \ldots, f_n\}$, we add the attribute `role="group"` to that node to communicate the presence of a labeling group. Next, we add the `aria-labelledby` attribute to the same nearest common ancestor node, and set the value of the attribute to the id attribute (whether existing or generated) of the mapped label t_i associated with the set $\{f_1, \ldots, f_n\}$. Finally, we note that the markup repair process can be executed at any time, whether on page load, or whenever the DOM of the form is updated in dynamic pages.

Implementation. We implemented the proposed approach using server-side JavaScript (Node). We used the Selenium WebDriver to instrument browsers and extract DOM information and computed attributes. To make the study replicable, we made available online a link to our data and tool [183], which we called AXEFORM (short for accessible forms). We used OpenCV [45] for image operations and Lpsolve [4] to formulate and solve the optimizations.

5.4 Evaluation

We conducted qualitative and quantitative studies to answer the following research questions:

RQ1 How accurate is the labeling associations inference?

RQ2 How safe is the labeling markup repair process?

RQ3 How scalable is the performance with the size of real-world web forms? Is it suitable for real-time usage?

In the following subsections, we discuss the details of the experiments that we designed to answer each research question, together with the results and discussions.

5.4.1 RQ1: Inference Accuracy

In this question, the objective is to assess how accurate is the inference of the labeling associations in a given web form. We recall that, as per the ARIA standard, a web form is accessible if it has markup that correctly captures the labels on the form. Accordingly, the most important evaluation question is to assess how accurate are the inferred form labels.

We evaluated this research question as follows. First, we collected 30 random subjects that were sourced from the Alexa top websites list [2]. The way we selected the evaluation subjects are as follows. To start, we get a random subject url from the pool. We then load that url in a browser in order to inspect the website. This inspection process is conducted manually, and its goal is to find a page on the website that contains a web form. For each website, we manually inspect its various sections and pages until a web form is found. If no web forms are found within five minutes of manual examination, the subject is skipped and a new random url is obtained from the pool. The only additional criterion we have on web forms is that they are reachable without requiring registration or payment. This criterion was made in order to simplify the process of collecting subjects. The final list of subjects is shown in Table 5.1, and the full urls are available online [183]. The final list of subjects covers a variety of tasks from many categories of websites (e.g. news, education, commerce), with a varying number of elements in forms, ranging from 17 up to 1552, which covers around two orders of magnitude of form sizes. This variety of topics and size ranges helps in having a more representative and generalizable evaluation. Subsequently, we feed the web form to our approach and obtain the output labeling associations. We then examine each generated association and classify the results into true positives, false positives, and false negatives. In false positives, the inferred labeling association for a given field does not match the visually perceived labeling. For instance, in Figure 6.1b, if the inferred labeling associates the selected radio button to the label ‘Yes‘, then this is a false positive labeling, because it does not match the correct visually perceived labeling, which is the ‘No‘ option. Next, in true positives, the inferred labeling association for a given field correctly matches the visually perceived labeling. For instance, in Figure 6.1b, if the inferred labeling associates the large text area input to the label ‘Message‘, then this is a true positive labeling. Finally, false negatives are cases in which no labeling associations were inferred for a field that should have been associated with a label. For instance, in Figure 6.1b, if no labels were associated to the first text field input, then this is a false negative, because there should have been a label (i.e., ‘Name‘).

Finally, in order to have a more thorough and informative evaluation, we include a baseline in our experiments. While we could simply present the evaluation from just the approach itself, this would not provide context as to how it would perform compared to other potential solutions, and therefore adding a baseline helps in making the evaluation more meaningful. However, as discussed in the introduction, existing works only test the accessibility of markups, but do not conduct any inferences or repairs of form labeling [256, 280]. They assert that accessibility attributes are not empty, while the proposed analysis in this chapter infers what *values* should be assigned to the form labeling attributes. Accordingly, no comparable tool exists that can be included in the evaluation, and therefore the next best option is to have a random selection process as the baseline. In this process, random elements from a given web form are selected. Next, a labeling association is generated to another randomly selected form element. This set of randomly selected elements and their associations is then taken to be the baseline.

Table 5.1: List of the 30 subjects used for evaluation.

Subject	**Form description**	**# elements in form**	**total # elements**
wikipedia.org	Page links search	45	518
opinionlab.com	Experience feedback	116	136
zoom.us	Live demo request	437	924
netflix.com	New titles suggestion	17	432
microsoft.com	Careers support ticket	86	444
dropbox.com	Business account inquiry	398	721
stackoverflow.com	Help center	106	743
etsy.com	Product design careers	68	340
zendesk.com	Customer service inquiry	428	1323
bing.com	Search customization	1552	1642
github.com	Account support request	229	283
intuit.com	Career events sign-up	326	1293
salesforce.com	Website quality feedback	102	764
indeed.com	Account registration	54	222
paypal.com	Search for jobs	144	410
coursera.org	Services inquiry	569	1210
glassdoor.com	Sales specialist contact	1536	1864
insiderintelligence.com	Subscription request	391	1019
formstack.com	Violations reporting	126	182
dailymail.co.uk	Message board help	45	986
squarespace.com	Press contact	33	725
elsevier.com	Advertising request	437	940
hootsuite.com	Community enrolment	61	560
flickr.com	Support ticket request	243	348
goodreads.com	Advertising inquiry	75	551
slack.com	Sales contact	461	1087
evernote.com	Teams products inquiry	298	730
udemy.com	Demo request	50	241
blackboard.com	Personalized experience	641	838
mailchimp.com	Report compliance issues	58	1209

Results and Discussion

Table 5.2 shows the results of evaluating the accuracy of inferring web form labeling. The table has two groups of columns, “Proposed approach” and “Baseline”, showing the accuracy of inference for both methods, respectively. The key outcome of this table is the F-1 measure, which is at 89% for the proposed approach. This indicates a rather effective inference process. Precision and recall were comparable, at 88% and 89%, respectively. Figure 5.3 shows a sample of the inferred labeling associations corresponding to the motivating example. Each entry represents a mapping from a particular field to a particular label.

Table 5.2: Evaluation of the inference accuracy of labeling associations.

	Proposed approach			Baseline		
Subject	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
wikipedia.org	3	0	0	0	5	1
opinionlab.com	6	2	0	0	11	1
zoom.us	12	0	2	0	13	4
netflix.com	3	0	2	1	4	3
microsoft.com	5	0	0	0	15	0
dropbox.com	9	0	1	0	7	4
stackoverflow.com	4	1	0	0	6	1
etsy.com	3	0	2	0	2	3
zendesk.com	6	0	0	0	9	1
bing.com	11	7	1	0	5	4
github.com	5	0	0	0	9	2
intuit.com	6	0	0	1	7	2
salesforce.com	3	1	1	0	8	1
indeed.com	5	2	1	0	6	2
paypal.com	4	0	0	1	3	1
coursera.org	12	0	1	0	6	5
glassdoor.com	7	0	3	0	5	3
insiderintelligence.com	7	3	0	0	4	6
formstack.com	6	1	0	0	3	5
dailymail.co.uk	8	0	1	1	4	3
squarespace.com	5	1	0	1	5	1
elsevier.com	15	1	2	0	6	10
hootsuite.com	5	0	0	0	4	1
flickr.com	6	1	2	0	3	6
goodreads.com	7	0	1	0	2	5
slack.com	10	2	2	0	8	4
evernote.com	6	1	0	0	4	3
udemy.com	7	0	0	1	3	3
blackboard.com	13	3	1	0	5	11
mailchimp.com	8	1	1	0	6	4
	Prec.	Rec.	F1	Prec.	Rec.	F1
	88.4%	89.6%	89.0%	3.3%	5.7%	4.1%

In order to further understand the limitations of the approach, we investigated the false positive and false negative cases. We identified a few trends. First, false positives occurred in forms that had visual prominence cues that did not follow the proposed optimization model (Section 5.3.2). In these cases, the forms had a high ratio of texts relative to fields and all the texts had similar visual prominence cues. This resulted in the optimization process yielding suboptimal labeling associations.

False positives also occurred in cases where, in addition to the aforementioned similarity in visual prominence, the visual layout was dense, which made it difficult to compensate for the layout density using visual prominence information.

As for the false negatives, the most predominant case was due to non-standard form elements. That is, these cases did not follow the normal practice of using input tags to indicate form fields. The most common example of this case is Captcha elements (e.g., "I'm a human" checkboxes), which are designed on purpose to avoid being detected by automated tools as form fields. Such elements are only styled to appear, for instance, as a checkbox when observed by sighted users. For these reasons, existing studies [188, 195] have also confirmed these hindrances of accessibility due to Captcha elements. The proposed approach is incapable of handling these cases. For future work, a potential solution to this problem might include formulating another set of visual cues to detect the missed cases, or exploring the use of a deep learning detector.

We also examined the effect of form size on the inference accuracy, as shown in Figure 5.4. The figure shows the average F1 score of inference for three different groups of form sizes, which are the 0 to 33, 33 to 66, and 66 to 100 percentile of form sizes as measured by the number of DOM elements in the form. First, we note that the first group had slightly lower inference accuracy compared to the second group. But we do know that the number of false positives have stayed the same, and therefore the lower accuracy can be attributed to the arithmetic of computing accuracy, where a single inference error in a form with a small number of elements would impact the accuracy more so than a single inference error among a large number of form elements. Second, we note that the lowest accuracy was for the last group (i.e., largest forms). This is due to the observation mentioned in the pre-

```
{ // associations.json
  // each entry is xpaths of field -> label

  ".../form/textarea[@id='bx_3978']":
      ".../form/div[1]/p", // <p>Message</p>

  ".../form/input[@id='vr_9481']":
      ".../form/section[1]/div[1]/p", // <p>Name</p>

  ".../form/select[@id='frq']":
      ".../form/section[2]/div[2]/p", // <p>How often ...
          </p>

  // remaining elements ...
}
```

Figure 5.3: Sample of the generated association decisions output (corresponds to Figure 6.1).

ceding paragraphs, where the larger forms tend to have a more dense visual layout, which reduced the quality of the optimization decisions as the cues became less discriminating due to the higher layout density. However, the observed reduction in accuracy is not that significant, and we emphasize that all the aforementioned observations are relative in nature with respect to the other size groups. Accordingly, the key observation of this evaluation is that the inference accuracy remains relatively stable across size ranges in Figure 5.4.

5.4.2 RQ2: Markup Safety

In the previous research question, we examined how accurate the labeling association inference is. Once that aspect has been evaluated, we need to examine whether the insertion or augmentation of the inferred associations into the DOM is safe. The rationale for evaluating this aspect is that we want to make sure that any markup repairs applied to a page does not cause any unintended or unaccounted for breakages or failures to the page.

We evaluated this research question as follows. First, we continued using the same test subjects used in the first research question. Each subject is then loaded in a browser and had their labeling associations inferred and the DOM repaired. To assess the safety, we check two different aspects. First, a visual check is made comparing images of the page before and after the repair. The rationale is to assess whether the generated markup would cause unintended breakage to the rendering of the page. Second, a functionality check is made. Here, we randomly manipulate the form (e.g., we select various radio options, expand different drop down menus) to assess whether the generated markup caused unintended breakage to the form. Breakages might occur, for instance, if the generated markups are incorrect, or break some element or attribute dependencies in the script of the page. Finally, we note down the outcome (i.e., pass or fail), and note down the number of failures.

Results and Discussion

Table 5.3 shows the results of evaluating the safety of the markup repair. For each subject, the columns show the outcome of the safety check and the number of failures. Subjects for which no failures were observed have '-' under the number of failures column. For most subjects, we did not observe any failures. For two of the subjects, as shown in Table 5.3, we did observe failures. In both these cases, the DOM repair process failed. The reason for these failures is locator breakages. This occurs because after the labeling association for a field is determined, locators are used locate the field in the DOM, then insert the markup repairs at that location. But in the case of the observed failures, the locators became stale, where locators that were previously valid become unusable. This occurred because the subject's DOM was highly dynamic and therefore the DOM has changed since the time the

Table 5.3: Evaluation of accessibility markup augmentation safety.

Subject	Outcome	# failures
paypal.com	fail	3
intuit.com	fail	2
remaining subjects	pass	-

locator was acquired. In this work we used DOM locators in the form xpath strings. The issue of stale locators and possible approaches to address them have been also reported in other web testing and analysis works [136, 152] and is still an open research problem. For future work, we plan to explore different possibilities to capture the locators and address these issues. One option would be devising a more robust timing strategy at which to capture the locators in order to minimize the probability of going stale. Another option would be using visual locators instead of DOM locators.

5.4.3 RQ3: Runtime Scalability

After evaluating the accuracy and safety aspects in the previous research question, this question examines the runtime performance (i.e., total time of execution). The rationale for evaluating this question is as follows. First, since a couple of aspects in the optimization formulation are combinatorial in nature, we wanted to assess whether or not this is going to cause any performance penalty, and more importantly how does the performance scale with the size of forms. Second, since the goal of this work is to make web forms accessible, it would be useful to know if the runtime performance is good enough if this approach were to be used for real-time repair by end users during their browsing activities.

We evaluated this research question as follows. For each subject, we record the number of DOM elements in the forms as a measure of form size. We are measuring the elements in the form since our approach only focuses on form regions within the page, and is therefore not impacted by the rest of the page. This will be used to measure how does the runtime scale with form size. Then, the total runtime is measured from the moment the subject is loaded until the final DOM markup is augmented.

Results and Discussion

Figure 5.5 shows the results of the runtime performance evaluation. The x-axis is semi-log and shows the number of DOM elements within the form. The y-axis is linear and shows the runtime in milliseconds.

The average runtime is 1667 ± 532 milliseconds. Minimum and maximum runtimes are 1044 and 2733 milliseconds, respectively. The data range of DOM elements covers around two orders of magnitude of form sizes, making it suitable for assessing scalability. The runtime scales roughly linearly with the size of the DOM, indicating good scalability with respect to form complexity. We recall that we are implementing the optimization in the Lpsolve [4] solver. While the solver is a good fit for the type of optimization we are conducting, it is known to be lacking in terms of runtime performance [166]. Accordingly, while the observed average performance of around 1.6 seconds is arguably suitable for run-time repair, the performance will likely improve further when more capable solvers are used, and therefore the current evaluation is a conservative estimate. The key outcome, however, remains the same, which is that the proposed web form analysis scales linearly with respect to form complexity. This provides some indication that the formulated optimization problem and constraints do not practically result in a combinatorial explosion that would cause computationally prohibitive analysis.

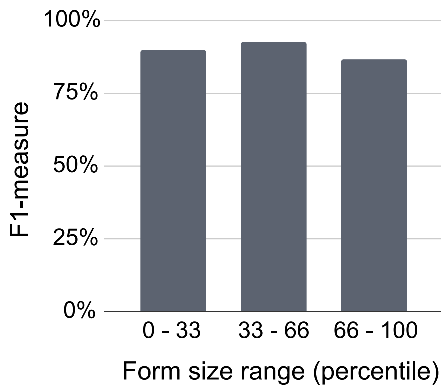


Figure 5.4: Comparison of labeling inference F1-measure for different form size ranges.

Threats to validity

In order to avoid any selection bias, we chose test subjects (i.e., web sites) randomly with the mentioned criteria in Section 5.4.1. The subjects cover diverse categories (e.g. news, education, commerce), with around two orders of magnitude of form sizes, ranging from 17 up to 1552 elements. This diversity of topics and sizes helps in ensuring subjects are representative of real-world scenarios, thereby mitigating the external validity of the study by making the results generalizable. To make the study replicable and mitigate researcher bias, we made available online a link to our tool implementation and evaluation data [183].

5.5 Related Work

A number of tools exist for repairing or improving a few accessibility issues [22, 280]. One common accessibility improvement approach is already in use in modern web browsers, such as allowing users with low-vision (e.g., mild vision problems, color blindness) to override the default CSS stylesheet into more contrasting colors in order to improve their visual perception of the page, or to zoom in to

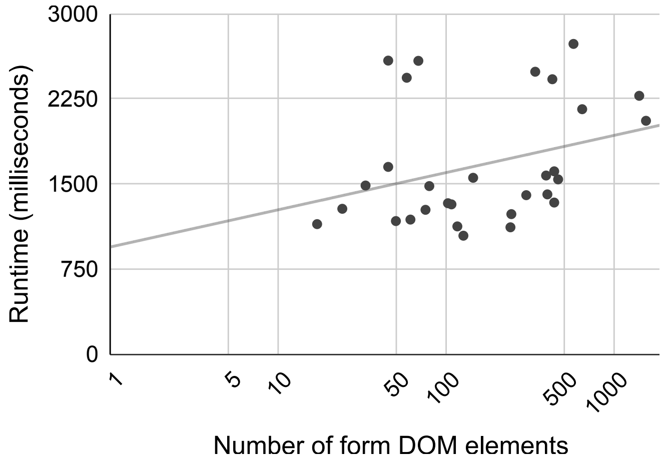


Figure 5.5: Evaluation of the runtime performance scalability for various sizes of input web forms. (Runtime in milliseconds. Form size in number of DOM elements in a form.)

the page in order to be able to better see its content. Another line of work aims at suggesting alternative texts for images to make them accessible [43, 167, 272], building on the abundant and extensive deep learning research on the task of image captioning [118]. Accessibility improvements have also been proposed based on restructuring of web page regions with the goal of prioritizing key sections so that they become more prominent and easier to access [12].

However, there is little to no work in the literature in terms of a software analysis approach that infers and repairs missing web form DOM labeling markup, which is one of the most common web accessibility errors [123]. This is the research problem that the presented approach addresses.

Another line of work focuses on accessibility testing [6, 256], which is generally based on performing syntactic checks that consist of basic rules, such as: any `ul` list element must contain non-empty `li` child elements, or no `input` elements must not be descendent of `a` elements. The goal of such checks is to provide quick and simple assertions that are easily automatable. Patil et al. [201] and Eler et al. [89] check for absence of predefined attributes in mobile applications, such as the absence of required alternative text for all user interface images. These are then flagged and reported to developers, allowing them to identify and fix these potential issues. Other tools [24, 107] perform similar syntax checks, differing mainly in what attributes are being checked. None of these works, however, perform repairs.

The majority of existing software accessibility related research come from a human factors research perspective, rather than from a software engineering or tooling perspective. For instance, one line of work [11, 18, 42, 85, 216, 238, 276] is based on analyzing existing websites (or categories of websites, such as educational or banking) by manually observing how non-sighted users would interact with them. The goal is to discover any issues that might surface during usage, and then come up with improved accessibility guidelines. The hope is that these guidelines will be later kept in mind while developing software. A related area of research aims to discover how the software development practices themselves can be improved such that it is more likely that the end product software is more accessible. Krainz et al. [141] explores how model-based development contributes to accessibility compliance. Sanchez et al. [225] and Bai et al. [27, 28] focus on agile software engineering practices and how do they impact the accessibility of the product.

Another line of research is concerned with generating inputs for testing purposes [10, 247]. These works have proposed techniques geared towards improving the generation, utility, or robustness of test inputs. One goal of this vast area of research is to create better sequences of test operations or test data with the goal of achieving a certain testing objective (e.g., increased coverage). [224, 240] adopt a strategy of randomly generating the inputs. In these cases, the focus is not mainly on the generated input data itself but on other aspects of testing, such as improving how the event space is traversed or how to better generate assertions, with the raw input data being randomly generated, or predefined by the user. Another line of work [20, 81] gives more attention to the raw input data. In these cases, specific types or formats of input values are used instead of randomly generated inputs. The intuition is that properly formatted and typed inputs allow better exploration and testing. The aforementioned works, however, are not related to accessibility testing or repair, nor aim at generating the required labeling associations to be able to perform accessibility repair.

Finally, a few existing works have explored the use of some visual analysis aspects for testing or analyzing web applications. Burg et al. [52] propose a program understanding tool that is geared towards front-end development projects and understanding how the user interface changes during interactions with the application. It allows a developer to select a certain element that they are interested in, and the tool tracks how the UI is changing with respect to that element, and tracks the corresponding code changes. Bajammal et al. [32] analyzes the UI of a web page in order to generate reusable components, based on patterns in design mockups or templates. Choudhary et al. [65] focuses on cross-browser compatibility, and proposes a technique that checks for difference between how a given web page is rendered across browsers. A given web app is loaded in two different browsers, and an image diff is performed to locate the areas where the two browsers differ, therefore helping the developer to identify issues that may not otherwise be easy to identify.

5.6 Conclusion

Filling web forms is a key activity when browsing the web. While this task is routine for sighted users, it presents a significant hurdle for non-sighted users if the form does not contain the required DOM accessibility labeling markup. This issue of missing labeling markup is one of the most common accessibility errors. However, when a non-sighted user is faced with missing form labeling, there are currently little to no options available to access that form. To this end, this chapter proposed a software analysis approach that automatically analyzes web forms and infers their labels to make them accessible. The approach first abstracts a given web form, then generates visual cues from the form. These cues are then used in an optimization model to solve for the form labeling associations. These are finally translated into standard ARIA accessibility markups and augmented into the DOM to repair the form and make it accessible. We evaluated our approach on 30 real-world subjects and assessed the accuracy of labeling inference, the safety of the DOM augmentation repairs, as well as the labeling performance. The results show an average F1-measure of 88.4% for label inference, and an average run-time of around 1.6 seconds.

Chapter 6

Generating web UI components

6.1 Introduction

The development of user interfaces (UIs) for web apps is often a manual and time consuming task.In a survey of more than 5,700 developers, 51% reported working on app UI design tasks on a daily basis [121], more so than other development tasks, which they tended to perform every few days. Another study also showed that an average of 48% of the code size of software is related to the user interface [190].

A common workflow for creating web user interfaces is *mockup based design* [192, 199]. In this approach, a graphic designer creates a rough illustration of the anticipated UI design, called the *mockup* or *wireframe*, usually through a graphic design software or a WYSIWYG editor. This mockup is then exported to HTML to be rendered in a browser. A web developer then examines the mockup and begins constructing web components for the app, which are nowadays implemented in one of the popular front-end frameworks such as ANGULAR [104] or REACT [128].

The main building block of UI design, and a cornerstone of these front-end frameworks, is the concept of *reusable components* [105, 129], which are a set of APIs and coding practices allowing reuse and encapsulation of repeated patterns on the front-end. Reusable components help improve modularity and maintainability, make the code more testable, and effectively remove duplication, by offloading the task of creating repetitive patterns to the web browser at runtime. Recent surveys show that using front-end frameworks is extensively popular among web developers. In one survey more than 92% of around 28,000 surveyed web developers stated that they use a framework rather than constructing UIs using pure HTML [243]. As a result, creating reusable components is often an essential element of building an app's front-end.

This component creation process can often be time consuming and tedious [210] in practice; it requires several manual steps, including the examination of the mockup, checking potential elements that may or may not be suitable for conversion to components, constructing a template for components that unifies repeated segments, adding placeholders for variable content, and refactoring the code to replace instances with instantiated components [210].

To the best of our knowledge, there has been little to no automated support in creating these reusable web components from mockups. Existing techniques help to manage mockups themselves, but do not generate any components. For instance, one set of approaches [207, 237] takes a mockup as input and converts its layout into a responsive code (e.g., through CSS) such that it is flexible to maintain the layout on different display sizes. Others [181] propose a tool that overlays the mockup as a transparency layer while implementing the UI, and performs a snapping-like functionality that aligns against various parts of the mockup.

In this chapter, we propose a technique, implemented in a tool, VIZMOD, to fill this gap by automatically generating reusable web components from mockups. Given a web mockup, our technique automatically identifies patterns on the UI, refactors the HTML code, and creates reusable components for popular front-end frameworks that are already familiar to developers such as REACT or ANGULAR. At the core of our approach is an unsupervised machine learning process for the detection of reusable UI patterns; we use features composed of a hybrid of information obtained from the Document Object Model (DOM) as well as the visual analysis of the UI.

We evaluate VIZMOD on five real-world web mockups by automatically identifying and transforming 120 component instances into 25 components. We also ask five experts to manually find patterns on the mockups and compare the output from our approach with the manually-identified patterns. Our approach is able to achieve 94% precision and 75% recall, on average, in correctly detecting reusable patterns in the UIs.

This chapter makes the following main contributions:

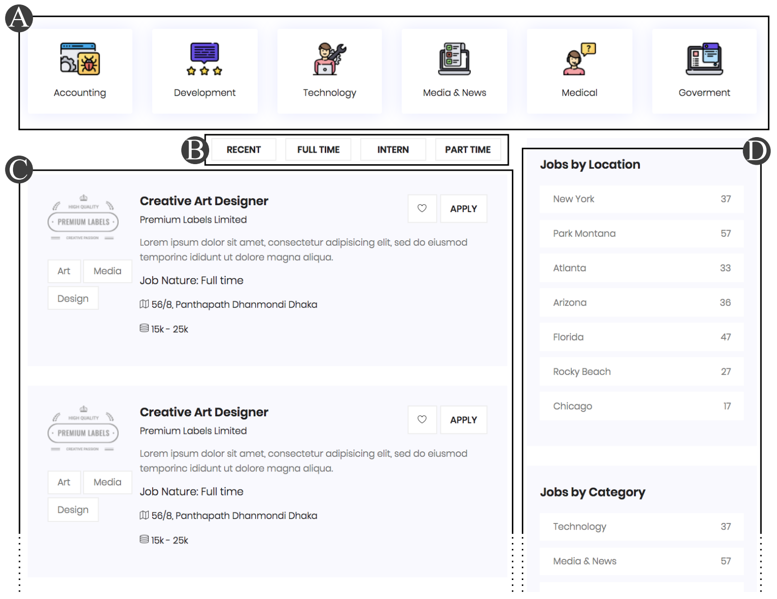
- A novel approach for automatically generating web components (e.g., REACT, ANGULAR) from mockups, which is the first to address this issue, to the best of our knowledge.
- An implementation of our approach, available in a tool called VIZMOD.
- A qualitative and quantitative evaluation of VIZMOD in terms of its accuracy and reusability of the generated components.

6.2 Motivating Example

Figure 6.1 illustrates a part of a sample UI mockup for a job hunting website. The mockup, often designed by a graphics designer in a team, provides a visual representation of what the UI of the web app is supposed to look like. The code corresponding to this mockup includes the HTML code, which defines the mockup's structure and content, and CSS code, which defines its style and presentation. This code is typically generated automatically using popular web UI editors (e.g., Muse, Dreamweaver, Visual Studio). The HTML and CSS code is interpreted by web browsers to render the UI.

Subsequently, a web developer oversees the creation of the final front-end code for the app. For the vast majority of developers, a major part of this process is the creation of reusable components [243]. Using components is key in improving modularity and maintainability and achieving the software engineering best practice of DRY (Don't Repeat Yourself). It is also an effective way to remove duplications in the app's code, which has been shown to be associated with increased error-proneness [131], maintenance effort [165], code instability [184], as well as higher hosting costs and rendering delays due to the transmission of redundant data. The utilization of reusable web components can help to address these issues.

For example, observe in Figure 6.1 that there are four groups of elements repeated in the mockup, denoted by 🅐, 🅑, 🅒 and 🅓. Notice that the repeated elements are not exactly similar; there are differences in terms of, for example, the text and images appearing within the elements. Nevertheless, the structure of the repeated elements in each group and their overall visual appearances are unquestionably repeating. Reptitions in UI are unavoidable and neccessary. In fact,



Creative Art Designer
Premium Labels Limited
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod temporinc ididunt ut dolore magna aliqua.
Job Nature: Full time
56/8, Panthapath Dhanmondi Dhaka
15k - 25k

Creative Art Designer
Premium Labels Limited
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod temporinc ididunt ut dolore magna aliqua.
Job Nature: Full time
56/8, Panthapath Dhanmondi Dhaka
15k - 25k

Jobs by Location

New York	37
Park Montana	57
Atlanta	33
Arizona	36
Florida	47
Rocky Beach	27
Chicago	17

Jobs by Category

Technology	37
Media & News	57

Figure 6.1: An example of a web UI mockup.

repetition is an important aspect of effective visual design [179], and is known as a *functional technique* to achieve appealing designs [86]. Research has shown that, when a *visual stimuli* is repeated, it is more likely to be accepted by people, a phenomenon called *repeated exposure* [157].

However, the process of creating reusable components is rather time consuming, requiring manual effort [210]. When analyzing repetitions in a mockup to construct components in one of the modern UI frameworks, developers often face the following challenges:

- They need to visually glance at the mockup and manually identify the patterns in the UI that can be potentially refactored to create a reusable component. For instance, for group A, they need to find all patterns in the page that represents components similar to the elements inside group A. The repetition might be spread across the web page, making the identification more challenging. The developer has to repeat this same process for other groups of components on the page, which quickly becomes a time consuming manual effort. Note that, this identification is not possible by only using existing code clone detection tools that support HTML code as input (e.g., NICAD [218]), due to several reasons:

1. These tools leave out the visual appearance of the elements and only work at the source code level, which is sub-optimal since there are several *inherent patterns* in HTML which do not necessarily represent a UI component. For example, HTML tables are declared using a `<table>` tag followed by a series of other tags, e.g., `<thead>`, `<tbody>`, `<colgroup>`, `<tr>`, and `<td>`, nested in a predefined hierarchy. The clone detector might mark all tables on the page for extraction, even if they do not visually constitute a reusable component in the UI. The same happens for several other elements, such as (un)ordered, description, and drop-down lists.
2. Clone detectors need to be configured properly in order to yield desirable clones. There is usually a large list of parameters and thresholds to tune, and finding an optimal configuration is a laborious task [263].
3. Clone detectors are not aware of the ultimate reason for detecting clones, e.g., there is no configuration that can force them to only identify clones that can be unified into a component template.

- The developer also needs to *unify* the patterns to construct a reusable component in a UI framework. This process needs careful investigation of repeated HTML, to identify how elements can be unified into one representative component, and which elements can be *parameterized* when there are differences. For example, in group Ⓐ, a developer would examine each button in the group, and determine which parts are repeated between the buttons, and which part is variable (e.g., the button icon and its label). The constructed component should resemble the exact hierarchy of the original repeated elements, or else the output of the resulting UI might differ from the original one.

- Moreover, to use the constructed component, the developer has to *instantiate* it in the places where the repeated elements originally appeared, with the appropriate parameters (e.g., original texts or images) to preserve the output of the mockup. For example, in group Ⓐ, the developer needs to refactor the original code and replace every occurrence of a button with a call to the button component, passing along arguments for the button label and its image.

To the best of our knowledge, there has been no techniques available to address the aforementioned issues and support developers in the generation of components.

6.3 Proposed Approach

Figure 6.2 shows an overview of our proposed approach to automatically generate modularized reusable UI components from mockups. The approach begins by retrieving the DOM of the web app's mockup. Next, a visual abstraction is performed to generate a normalized and abstract representation of the web app's UI layout. This transforms the mockup into a set of visual elements (VEs) on which further analysis is conducted. The approach then performs a dynamic grouping of visual elements, to identify subtrees which correspond to potential instances of a UI component. This grouping is used in the next step, where an unsupervised machine learning technique applied on the potential UI component instances iden-

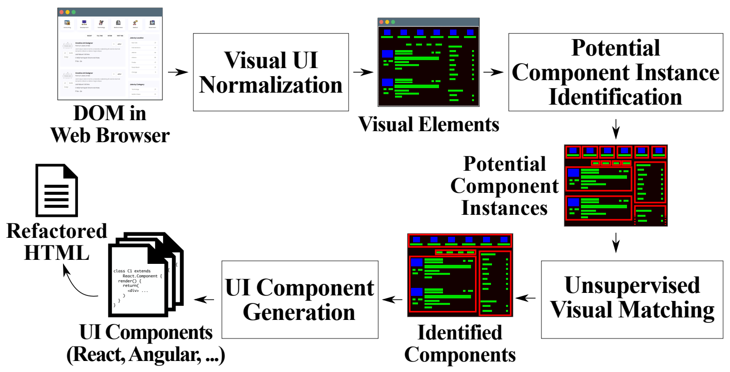


Figure 6.2: Overview of the proposed approach.

tifies UI components. Finally, the actual code for the UI components is generated by refactoring the original HTML code.

In the following subsections, we describe each step of the proposed approach and illustrate some of their major components and analysis procedures.

6.3.1 Definitions

Before we proceed to describe the details of the proposed approach, we begin by declaring a few important definitions that are used throughout the chapter.

Definition 3 (UI Component) *A UI component $c_E = \langle n, N \rangle$ for a repeated group of UI element trees E in a web application is a tree structure rooted at $n \in N$, where $N = T \cup P$ is a set of abstract user interface elements. The component includes the* template *T and the* placeholders *P. The template of the UI component denotes the nodes which do not change wherever the component is used (i.e.,* instantiated*), while the placeholders captures the changed nodes, whether partially or fully changed, as explained in Section 6.3.5.*

In this chapter, we use the terms *UI component* and *component* interchangeably.

Definition 4 (Component Instance) *A component instance $i = \langle c_E, f \rangle$ is a concrete and specific instantiation of a UI component c_E. Component instances share the template part with other instances of the same component, but differ in the placeholder parts. The function $f : P \to V$ assigns values $v \in V$ to the placeholders $p \in P$ of c_E.*

Definition 5 (Potential Instance) *A potential instance is a subtree of the* DOM *constructed for a web application's user interface, representing a concrete UI element tree that is* likely *to form a component instance, but may not be so.*

Potential instances are processed at multiple stages of the proposed approach until they are either discarded or associated with a component.

6.3.2 Visual UI Normalization

In the first step of the approach, we take as input the DOM of the mockup after it is loaded and rendered in a browser, and perform a *visual normalization* that transforms the DOM into a set of *visual elements*. The goal of this step is to normalize the visual presentation of a web user interface into a set of abstract elements that signify the salient features of the page from a visual perspective, which may represent potential component instances. The intuition behind this is that normalization and abstraction can be helpful to achieve our goal of detecting reusable patterns, since the exact and minute details are less relevant when identifying repeated regions of a web page. Furthermore, component instances are generally different from each other in some aspects, while they still have similar overall visual appearance. This normalization step enables obtaining a big picture to identify these potential similarities.

The visual normalization is achieved as follows. First, we extract from the DOM a set of nodes that represent visual content of the UI, and we refer to each of these as *visual elements*. We define two main types of visual elements: textual and graphical (image). The extraction of text content is achieved by traversing text nodes of the DOM. More specifically:

$$\Gamma_T := \{E(node) : node \in DOM_R \wedge \qquad node.hasTextContent\} \tag{6.1}$$

where Γ_T is the set of all visual elements that represent text in the UI, DOM_R is the rendered DOM in the web browser, and $E(node)$ maps the node to the corresponding element. The predicate $hasTextContent$ examines whether there is a text associated with the node, and covers two possibilities: non-empty nodes of type `#TEXT`, representing string literals in DOM_R, and nodes of `input` elements that have an associated text value (e.g., buttons or lists). Subsequently, we perform another extraction for image content. We define this as follows:

$$\Gamma_I := \{E(node) : node \in DOM_R \wedge \qquad node.hasImageContent\} \tag{6.2}$$

where Γ_I is the set of all visual elements that represent images. As in the previous case, the predicate $hasImageContent$ examines if there is an image associated with the node. This again has two possibilities: nodes of `<img>` elements and non-img nodes with a non-null background image attribute.

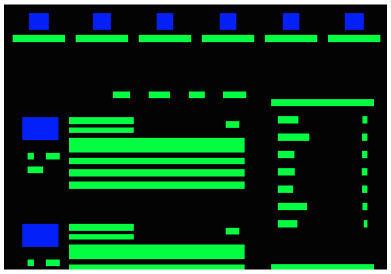


Figure 6.3: The result of the visual UI normalization stage (as applied to the motivating example of Figure 6.1). Best viewed on a color display.

Subsequently, we use the set of all visual elements to construct the normalized UI:

$$\mathrm{UI}_N = V(\Gamma_I \cup \Gamma_T) \tag{6.3}$$

where UI_N is the resultant normalized UI and V is a visual projection operation that generates an image from the union of visual elements. This is achieved as follows. First, we begin by collecting the final *computed* properties of each element, when rendered in the web browser. These properties represent the final state of elements after the propagation of all changes and events. The properties we collect are the size, location, and z-orders of these elements. Next, we assign different colors to each class of visual elements. We assign green for elements in Γ_T, and blue for elements in Γ_I. While any arbitrary colors could have been chosen, we chose these two colors in order to facilitate faster visual analysis in subsequent steps, since these two are typically represented in separate color channels. Figure 6.3 illustrates an example of the output generated from this visual normalization step. As can be observed, the minute details of the page are abstracted away while the main and essential structure of the UI is accentuated.

6.3.3 Potential Instance Identification

The result of the previous step consists of only a set of visual elements. These visual elements on their own do not *necessarily* represent reusable repetitive UI patterns. The goal of this step is to transform the set of individual visual elements into a set of *potential* reusable UI component instances. A potential instance is a region of the UI that has been determined to be likely repeated somewhere else in the UI. That is, it is a region where, using all of the aforementioned steps, the initial analysis have indicated that this instance would likely be a good candidate to be combined with other repetitions and refactored into a single component. These potential component instances will be further checked and analyzed in the subsequent steps in order to generate a final set of components.

Identifying potential component instances can be an intricate decision since there are multiple levels of hierarchy that can be considered. For example, consider group Ⓐ in Figure 6.1. Notice how the icons in that group would constitute repeated elements. The same is true for the text labels under the icons. Yet another repetition pattern is taking the icon and text as one component that is repeated multiple times. Accordingly, in order to identify potential component instances, we propose an approach that aims to maximize two complementary aspects, namely, the number of repetitions of a component, and the amount of repetitions encapsulated *within* each component instance (i.e., repetitions of the *same* potential instances). We refer to this combination of aspects as the *modularization potential*, where a high value of modularization potential indicates a potentially more reusable UI component. Our goal is therefore to utilize this modularization potential to optimize a set of potential instances, Ψ:

$$\Psi := \underset{C}{\operatorname{argmax}} \prod_{c_i \in C} \left| \{ c_i(\gamma_T, \gamma_I) : \gamma_T, \gamma_I \subset \mathrm{UI}_N \} \right| \tag{6.4}$$

where C is the set of all component instances, c_i is a potential instance, and the optimized function is the modularization potential. This optimization yields a global optimum set of potential instances between two extremes. At one end of the spectrum, each visual element represents a component of its own. This yields a sub-optimal component set that has low modularization potential because of a lack of repetitions. For this case, the modularization potential in eq. (6.4) yields a score of 1 since each component encapsulates only a single element. At the other end of the spectrum, one might theoretically consider the *entire* collection of visual elements to represent a single component that is repeated only once. This results in a score equal to N, the number of total visual elements, in eq. (6.4). Ψ, on the other hand, represents a global optimum between the aforementioned two extremes. Ψ captures a set of potential component instances that aims for *both* a large number of components, and for an instance that in itself has a large number of UI repetitions. The subsequent steps of the approach will therefore only use Ψ for further analyses and final generation of components.

We now describe the implementation for generating Ψ. Figure 6.4 shows an illustration of this process. First, we obtain DOM locators (e.g., XPATH expressions) for each of the visual elements. Next, starting from these locators as leaf nodes, we iteratively build a tree from the bottom up (as shown in Figure 6.4), adding the DOM parent of every tree node with each iteration. At each iteration, we calculate the modularizaton potential of eq. (6.4), with every node's subtree representing a

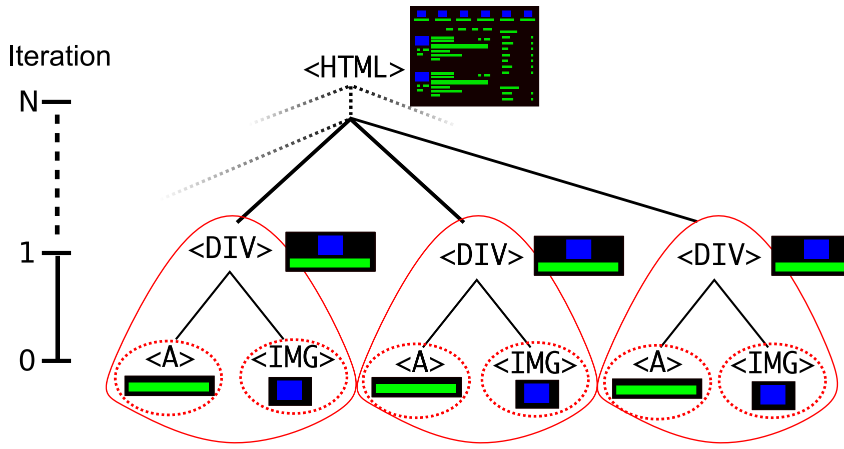


Figure 6.4: Illustration of the potential instance identification stage. Each iteration considers a different group of potential instances before selecting an optimum set.

potential instance c_i. The potential instances are illustrated using the red outlines in Figure 6.4. Note how at the very first iteration, each potential instance is simply the visual element itself. In the next iteration, the potential instances grow larger to include more visual elements as shown by the larger red outlines at iteration 1. Finally, the iteration that yields the maximal modularization is reported as the Ψ set and passed to the subsequent stage.

6.3.4 Unsupervised Visual Matching

The output of the previous step is a set of *potential* component instances that maximizes the modularization potential out of many alternative sets of instances. However, these are only *potential* instances that may or may not actually belong to a component. In other words, there is still no information as to which subgroup of potential instances do indeed belong together and constitute a reusable component, versus other potential instances that are simply visual elements that do not represent repetitive reusable components. In this stage, we process the set of *potential* component instances and reduce it into a final set of components.

In order to create the final components, we propose an approach that visually examines potential instances and combines them into components via unsupervised machine learning. The intuition behind adopting this approach is that if potential instances match with other potential instances, the "potential" qualifier can be dropped from these instances and they would be recognized as constituting a component together. In this approach, we use a clustering mechanism to create components in order to facilitate robust matching of potential instances.

We now describe the details of the process. First, we obtain the screenshot image of the visual elements per potential instance. This results in one image (containing all visual elements) for each potential instance. Next, for each potential instance image, we extract a feature vector. We compute the feature vector using a *vectorized pixel histogram*, which is a process that captures a summary of the overall content in the instance image. However, unlike typical approaches from the machine vision literature [162, 164] where a binning parameter (a parameter for categorizing pixels) is required, we generate the vectorized histogram without requiring this parameter. Instead, due to the nature of visual normalization that we have proposed, only two categories need to be considered: one for text visual elements, and another for image visual elements. Therefore, we finally end up with a feature vector for each potential instance. Subsequently, we compute the cosine distance between each pair $\mathbf{I}_i$, $\mathbf{I}_j$ of potential instances:

$$D_{i,j} = 1 - \frac{\mathbf{I}_i \cdot \mathbf{I}_j}{||\mathbf{I}_i||\,||\mathbf{I}_j||} \tag{6.5}$$

Next, we perform an unsupervised clustering process. The selection of an appropriate clustering is of paramount importance due to a couple of challenges. First, the clustering can be challenging due to the wide range of possibilities of arrangements and structures of component instances. In other words, there is potentially a large range of *inter-* and *intra-*component variations. This makes it difficult to use hierarchical clustering, for instance, due to its very high sensitivity to outliers and therefore would be a poor choice for handling large component variations, and also due to its high dependence on order of data, which can make it less effective for detecting instances far way from each other. Furthermore, performing a cut on the clustered hierarchies often requires specifying the number of clusters or some other parameter, which can be difficult and brittle to specify. Density-based algorithms (e.g., DBSCAN) would not be effective either, as they would have difficulty handling the *variable* densities present between potential clusters of instances. Accordingly, we opted for a technique that can be flexible enough to correctly identify such variations and be able to better recognize the final components. To do this, we select a method that performs variable-density clustering with a hierarchy of densities [56]. The hierarchy of variable-densities allows the method to automatically detect stable clusters in a parameter-free fashion. More importantly, the method is built to handle varying-densities, which becomes very important when handling the potentially large range of inter- and intra-component variations.

Once the components have been identified through unsupervised visual matching, we extract the corresponding locator in the DOM (e.g., XPATHs) per instance. The final result is a superset of component instance locator sets. This superset is passed on to the next step in order to combine the component instances into final components.

6.3.5 UI Component Generation

We propose an algorithm that unifies the UI component instances identified in the previous steps into a component implemented using a web framework (e.g., REACT, ANGULAR, HTML Web Components [189]). However, instead of directly generating the framework-specific code for components, we opt for constructing an *intermediate model* that effectively represents components at a higher level of abstraction. This allows building different *translation strategies* for generating the actual code for different frameworks from the same model, with the added benefit of remaining agnostic to the specific details of a particular framework. Our implementation supports the REACT [128] translation strategy, which is the preferred framework for a significant number of developers in practice [242, 243]. We first define the terms used in this step.

Definition 6 (Mapping Nodes Set) *Let $T = \{t_1 \dots t_n\}$ be the list of* DOM *subtrees for n instances of a UI component identified by the previous phases of the approach. A set $D = \{d_1 \in t_1, d_2 \in t_2 \dots d_n \in t_n\}$ of* DOM *nodes corresponding to T is a Mapping Nodes Set, when every pair (d_i, d_j) of* DOM *nodes belonging to D are* ***mapping****.*

Definition 7 (Mapping) *Two* DOM *nodes d_i and d_j are mapping (denoted as $d_i \leftrightsquigarrow d_j$) when:*

- *Both d_i and d_j are root nodes of their trees, or*
- *d_i and d_j are not root nodes, and*

– *$d_i.parent.tag = d_j.parent.tag$, and*

– *$d_i.parent \leftrightsquigarrow d_j.parent$, and*

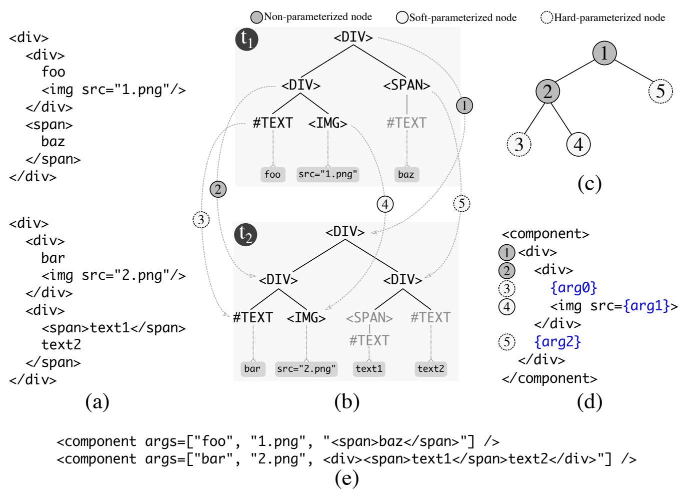
– *d_i and d_j have the same child index (e.g., they are both the first child of their parents).*

Definition 8 (Component Intermediate Model) *The Component Intermediate Model is a rooted, ordered tree in which each node corresponds to a Mapping Nodes Set. The hierarchy of this tree follows the mapping* DOM *nodes' hierarchy.*

Example. Figure 6.5(a) depicts the HTML code snippets corresponding to two identified UI component instances. The corresponding DOM subtrees, and the constructed Component Intermediate Model for these subtrees are respectively shown in Figure 6.5(b) and (c). The connected DOM nodes with dotted arrows form Mapping Nodes Sets. Notice that non-mapping DOM nodes do not form a node in the intermediate model. This model can be translated to a REACT-like component similar to what is shown in Figure 6.5(d). Finally, the generated component is instantiated two times in the refactored HTML code to replace the originally-repeated DOM nodes. The calls to the component can look like what is shown in Figure 6.5(e).

When generating the actual framework code, each model node results into a DOM node in the framework component (as depicted in Figure 6.5(d)), which essentially *unify* the nodes in the Mapping Nodes Set to remove duplication. There are three possibilities for framework component DOM nodes:

• When all pairs of DOM nodes in a Mapping Nodes Set have the same tag and identical attribute values, they can be unified in one DOM node of the same tag.



```
<div>
  <div>
    foo
    <img src="1.png"/>
  </div>
  <span>
    baz
  </span>
</div>

<div>
  <div>
    bar
    <img src="2.png"/>
  </div>
  <div>
    <span>text1</span>
    text2
  </span>
</div>
```

```
<component>
1 <div>
2   <div>
3     {arg0}
4     <img src={arg1}>
    </div>
5   {arg2}
  </div>
</component>
```

```
<component args=["foo", "1.png", "<span>baz</span>"] />
<component args=["bar", "2.png", <div><span>text1</span>text2</div>"] />
```

Figure 6.5: (a) Initial HTML code fragments. (b) Corresponding DOM subtrees. (c) The constructed Component Intermediate Model. (d) The final generated UI component. (e) The *calls* to the generated UI component.

For example, the two `<div>` nodes in Figure 6.5 corresponding to model node ❶ form a `<div>` node in the component.

- A pair of DOM nodes in a Mapping Nodes Set which have different tag names cannot be unified into one DOM node in the component (e.g., `<span>` and `<div>` corresponding to model node ❺ in Figure 6.5). Similar is two text nodes with different content (e.g., the `foo` and `bar` corresponding to the model node ❸ in Figure 6.5). In such cases, the DOM nodes (and the whole subtree rooted at them) should be *hard-parameterized* in the resulting component, i.e., a *placeholder* should be created. The original parameterized DOM nodes are later passed as *arguments* when instantiating the component to recreate the original DOM hierarchy.

- A pair of DOM nodes in a Mapping Nodes Set that have the same tag name but different values for one of their attributes might be unifiable into a DOM node via *soft parameterization*, where the differing attribute values are parameterized (e.g., the `<img>` tags corresponding to model node ❹ in Figure 6.5, with parameterized `src` attribute values). This can be done only if the used framework supports parameterizing attribute values. Otherwise, the parameterization should be done as if it was a hard parameterization.

The intermediate model construction and refactoring algorithm. The inputs of Algorithm 1 are the original mockup HTML, the list of component instance DOM subtrees, and the translation strategy. The output is the refactored HTML wherein duplication is removed using the UI components.

Algorithm 1 starts by constructing an empty model (line 1), and an exclusion list (`coveredNodes` in line 2) that contains the original DOM nodes of the component instances which are already covered by the algorithm (e.g., a model node has been created for them), so that they are skipped in future iterations. To construct the intermediate model, the algorithm chooses the DOM subtree of one of the component instances (i.e., the *template subtree*) to follow its hierarchy. The template subtree is the one with the smallest number of DOM nodes, chosen in line 3. This is because the intermediate model cannot have more DOM nodes than the smallest subtree, as it resembles the *intersection* of the component instances' DOM subtrees. The algorithm loops over all the uncovered template subtree's DOM nodes, following the subtree's breadth-first traversal order (lines 5 to 28). Each template DOM node is compared to other DOM nodes of its Mapping Nodes Set (identified according to **??** 6 in line 7) using the `compare()` function (line 10), which returns the type of parameterization needed to unify two given DOM nodes, and `NULL` if no parameterization is needed. Note that, even if one node in a Mapping Nodes Set

Algorithm 1 Component Intermediate Model Generation

```
Input: The original DOM of the mockup ($DOM_{original}$), UI component instances DOM subtrees ($subtrees$), UI component translation strategy ($strategy$)
Output: The new DOM after refactoring ($DOM_{refactored}$)
 1: $model \leftarrow$ CONSTRUCTEMPTYINTERMEDIATEMODEL()
 2: $coveredNodes \leftarrow \varnothing$
 3: $templateTree \leftarrow$ GETSMALLESTTREE($subtrees$)
 4: $templateNodes \leftarrow$ BFS($templateTree$)
 5: for $templateNode \in templateNodes \setminus coveredNodes$ do
 6:     $coveredNodes \leftarrow coveredNodes \cup \{templateNode\}$
 7:     $mappedNodes \leftarrow$ GETMAPPEDNODESSET($templateNode, subtrees$)
 8:     $parameterization \leftarrow$ NULL
 9:     for $currentNode \in mappedNodes \setminus coveredNodes$ do
10:         $parameterization \leftarrow$ COMPARE($templateNode, currentNode$)
11:         if $parameterization \neq$ NULL then
12:             break
13:         end if
14:     end for
15:     $parent \leftarrow model$.GETMODELNODEFOR($templateNode.parent$)
16:     if $parameterization \neq$ NULL then
17:         if $parameterization =$ SOFT_PARAMETERIZATION
                    $\wedge strategy$.SUPPORTSATTRIBUTEPARAMETERS() then
18:             $model$.ADDSOFTPARAMNODE($parent, mappedNedesSet$)
19:             $coveredNodes \leftarrow coveredNodes \cup mappedNodes$
20:         else
21:             $model$.ADDHARDPARAMNODE($parent, mappedNedesSet$)
22:             $coveredNodes \leftarrow coveredNodes \cup$
                        GETALLSUBTREENODES($mappedNodes$)
23:         end if
24:     else
25:         $model$.ADDNONPARAMNODE($parent, mappedNedesSet$)
26:         $coveredNodes \leftarrow coveredNodes \cup mappedNodes$
27:     end if
28: end for
29: $DOM_{refactored} \leftarrow strategy$.REFACTOR($DOM_{original}, model$)
```

should be parameterized when compared to the template DOM node, the resulting model node will be either hard- or soft-parameterized, thus comparing other nodes of Mapping Nodes Set is not required (line 12).

The intuition behind comparing nodes in the breadth-first order is that, across the component instances' DOM subtrees, it is more likely that the the inner nodes (which define the structure of the final UI component) are similar, while the leaf nodes (texts, images) are more probable to differ. The inner nodes are thus compared before leaves, also facilitating the identification of Mapping Nodes Set based on **??** 6, as the nodes' child indices follow the BFS traversal order.

The algorithm then continues to add a model node for each Mapping Nodes Set (lines 15 to 27). First, the model node that has been created for the template DOM node's parent (in the previous runs of the loop) is retrieved from the model (line 15), to which the new model nodes will be added as children. This effectively allows the model to preserve the original hierarchy of the instances' DOM subtrees. If the model is empty, the new model node will form the model's root. The subsequent lines of the algorithm add the new model node based on the parameterization type. In each step, the DOM nodes in the Mapping Nodes Set for which a model node is created are added to the `coveredNodes` to be skipped in the next iterations. As mentioned, in case of a hard-parameterized model node, all the DOM nodes belonging to the subtrees rooted under the corresponding mapping DOM nodes should be marked to be skipped (e.g., node ❺ in Figure 6.5).

Finally, the actual refactoring is conducted using the constructed Component Intermediate Model (line 29). The details of the refactoring are built-in the translation strategy, which can be implemented virtually for any UI framework of interest.

Implementation.

We implemented the proposed approach in a tool called VIZMOD [183] (short for **Vis**ual **Mod**ularizer). VIZMOD is implemented in Java and Python 3. We use the Selenium web driver to view the mockup and extract DOM trees and their relevant computed properties. For clustering, we use the implementation provided by Campello et. al. [56] and the `numpy` [261] library for mathematical and numerical functions.

6.4 Evaluation

To evaluate VIZMOD, we conducted qualitative and quantitative studies aiming at answering the following research questions:

RQ1 Are the refactorings by VIZMOD's component generation correct?

RQ2 How effective is VIZMOD in identifying UI components compared to manual examination by web developers?

RQ3 How much code reusability can be achieved through the proposed refactorings?

In the following subsections, we discuss the details of the experiments that we designed to answer each research question, together with the results.

6.4.1 RQ1: Correctness of Component Generation Refactorings

Study Design

For the proposed componentization applied on HTML to be safe, the main criterion is that the original and the refactored HTMLs must result into the same DOM tree landed into the users' web browsers. Consequently, to devise a technique that can automatically assess the safety of the applied transformations, we relied on the equivalence of the DOM subtrees rendered in the web browser, before and after refactoring. If the DOM trees are the same, given that our refactorings do not change any CSS style rules, the resulting presentation semantics of the HTML files remain intact.

To automate this process, we serialized the final DOM trees rendered in the browser to the pretty-printed HTML code and compared them pre- and postrefactoring. This allows a fast comparison of the structure of the DOM trees. We normalized the DOM trees by removing text nodes which are empty or contain only white spaces. This is done because REACT interprets these nodes differently [130, 209] compared to the standard HTML specifications.

Results and Discussion

Using the aforementioned technique, we compared the DOM subtrees of the UI component instances before and after refactoring for the 120 UI component instances (i.e., 25 UI components) identified by VIZMOD. The tests has passed for all subjects, indicating that the refactorings introduced by component generation do preserve the DOM trees, and as a result, the transformations are safe to apply.

6.4.2 RQ2: UI Component Identification

Study Design

We asked independent expert web developers to participate in a qualitative study, with the goal of understanding what they would identify as being a component pattern in a web UI. With this study, we aim at evaluating the (dis)agreement between the proposed approach and expert developers in terms of identifying the UI components.

Subject Systems. We searched the Internet to find mockups suitable for this study (using keywords like "web mockups", "web templates", "front-end templates"). Our selection criteria for choosing mockups were:

Table 6.1: Subjects' descriptive statics

Subject#	Body size (KB)	#DOM nodes	CSS size (KB)[†]
1	18	754	323
2	20	915	279
3	33	1,990	350
4	24	1,226	330
5	49	1,065	254

[†] Some mockups use CSS frameworks (e.g., Bootstrap). This corresponds to the total CSS size, including the frameworks.

- They should be non-trivial, both visually and code-wise (i.e., HTML and CSS). Note in Table 6.1 that the mockups are indeed complex, in terms of the number of DOM nodes and CSS code size.

- The number of subjects should be small and manageable enough so that we can ask participants to highlight potential components in *all* of them, without causing too much burden, mental fatigue, or boredom on them which can negatively distort the study.

- They should only represent the UI front-end, i.e., without back-end or front-end business logic or functionalities.

Based on these criteria we chose five mockups for our evaluation. We use the same mockups in all the evaluation experiments. Table 6.1 shows descriptive statistics for them.

Participants. We emailed developers who have worked in local businesses or research labs and asked them to voluntarily participate in our study. We attached a zip package containing our subject systems together with a link to a post-study questionnaire aimed at collecting information about the participants' demographics (e.g., number of years of experience in software engineering in general and in web development in particular, and their self-assessment on web application development skills). We asked them to manually highlight repetitions on the UI of each subject (by drawing a rectangle on each repetition) and send the results back to us.

Accordingly, we emailed 10 developers and informed them of the study, and asked them to also pass it on to their contacts. We received responses from a total of five developers. Table 6.2 shows participants' demographic information. As it is shown, all the participants were quite experienced in web development, as measured by the years of software and web development experience and their own self assessment.

Comparison with Developers. For each subject, we compared the components highlighted by the experts with those components that VIZMOD automatically identified as UI components. In particular, when more than half of the experts highlighted a pattern on the mockup as repeated, we assume the majority is correct and consider it as a UI component that our technique should be able to identify. The performance of VIZMOD is then determined using the well-known *precision* and *recall* measures. A *true positive* for the approach is defined as a UI component that has been manually identified by more than half of the experts (in our case, three or more participants). A *false positive*, on the other hand, is a UI component that is reported by the approach, yet less than half of the experts have identified it.

Table 6.2: Demographics of the Participants

Participant#	**SW dev. (#Years)**	**Web dev. (#Years)**	**Web dev. self assessment (1–5, 5=Highly Expert)**
1	10	5	4
2	8	2	4
3	3	3	4
4	11	3	3
5	9	8	5

Finally, a *false negative* of the approach is a UI component that is reported by more than half of the experts, but the approach could not identify.

Results and Discussion

Table 6.3 shows the results of comparing the UI patterns identified by our approach to the UI patterns identified by participating developers in our experiment. The table shows the values for true positives, false positives, false negatives, and finally precision and recall. The values for recall range between 74% and 100%. We examined the subjects at the lower end of the range to investigate further. Almost all the components that were missed by our approach had many elements that had animations or moving sub-elements (e.g., a carousel that changes every few seconds). Our technique was not designed with animations in mind. Capturing and analyzing animations can be challenging, due to difficulties in keeping track of changes over time and deciding which time instant to take as representative. This might be a possible venue for future work.

As for the precision, we further examined the nature of false positives in order to better understand the performance. Following this examination, we identified another variable while performing the comparison with participants: the *potentially missed opportunities*. We define missed opportunities as those patterns that were reported by as few as one developer (but not the majority), *as well as* our tool. The reason for introducing this variable is that, by manually examining the false positives, we noticed that there were a few potential opportunities that were missed by the majority of developers. We postulate a number of possible causes as to why such patterns were not reported by the majority of participants:

Table 6.3: Comparison of automatically identified components to manually-identified ones by developers

Subject	**#Identified Refactoring Opportunities**		**FN**	**TP**	**FP**	**Precision**	**Recall**	**Considering PMO**[†]		
	#Components	**#Component Instances**						**#PMO**	**Precision**	**Recall**
1	5	29	0	19	9	67.9%	100.0%	8	96.4%	77.1%
2	5	27	0	15	2	88.2%	100.0%	2	100.0%	77.3%
3	5	26	6	17	5	77.3%	73.9%	0	77.3%	68.0%
4	6	21	3	9	12	42.9%	75.0%	12	100.0%	84.0%
5	4	17	3	13	4	76.5%	81.3%	3	94.1%	69.6%
Avg.	**5**	**22**				**70.5%**	**86.0%**	**5**	**93.6%**	**75.2%**

[†] PMO = Potentially Missed Opportunities.

- Some repeating components were laid out far away from each other (e.g., at the very top and very bottom of the page). This often makes it difficult for human developers to remember patterns that are not immediately visible within the same view, especially if there are lots of patterns that they have to keep track of. The human brain has been shown to have a short-term memory capacity of only around 3 to 7 objects at a time [72]. This fact, coupled with patterns that are far away from each other and interlaced with multiple other patterns, can cause humans to miss some patterns. Our approach, however, is agnostic to where the pattern is located, and is able to recognize matching patterns from far ends of a web UI just as easily as patterns immediately next to each other.
- Some components included images or icons that were designed to be faint or barely visible due to artistic reasons. Such icons, especially when present close to very vibrant and large repeating components, are often skipped potentially due to the visual attention in the brain being directed at the larger clearer patterns. However, due to the visual normalization adopted in our approach, such artistic choice do not make any difference and the pattern is recognizable regardless of how visually pronounced it is.

6.4.3 RQ3: Code Reusability

Study Design

We now proceed to determine how much code reusability can be attained with the components generated by the approach. For each test subject, we compare the size of the HTML code of the mockups before and after refactoring as a measure of how much reusability has been achieved.

Let T_r be the set of DOM subtrees corresponding to the UI component instances which are going to be removed by a refactoring operation, r, from the original HTML. The refactoring r adds the necessary code which unifies T_r subtrees into a UI component u_r to the original HTML code. Moreover, r replaces T_r subtrees with a set C_r of calls to instantiate u_r. Accordingly, the size reduction SR_r for the refactoring r is computed as:

$$SR_r = \sum_{t \in T_r} sizeOf(t) - sizeOf(u_r) + \sum_{c \in C_r} sizeOf(c) \tag{6.6}$$

where t is a component instance, c is a component instantiation call, and $sizeOf(x)$ is the number of bytes corresponding to x when serialized to HTML. In an HTML mockup, there might be several sets of component instances (i.e., several UI components might be created). The overall size reduction SR, which is achieved by applying the set R of all refactoring opportunities found in a mockup, is calculated as $SR = \sum\limits_{r \in R} SR_r$.

We calculate the size reduction in two different ways: 1) based on an implementation using a UI framework (which we have chosen to be REACT), and 2) based on the representation contained in the Component Intermediate Model. This is because each UI framework (e.g., REACT, ANGULAR) has its own syntax and idiomatic mechanisms for creating UI components and instantiating them. As a result, the actual size reduction would be different depending on which, and how, a UI framework is used. All UI frameworks, however, follow the same basic principle: the set of DOM nodes that can be unified into single DOM nodes form a *template* for the UI component, while other nodes form the *parameters* (i.e., *placeholders*) in the UI component. These placeholders are filled with the *arguments* passed when calling the UI framework. As a result, calculating the size reduction based on the nodes and arguments identified when constructing the Component Intermediate Model allows a more accurate determination of how the algorithm *intrinsically* performs in terms of code reusability, regardless of the differences between the many possible UI frameworks that can be used.

Moreover, when using a third-party UI framework, it is usually required that the framework's JAVASCRIPT library code is imported at the client-side so that the web browser is able to render the UI, potentially increasing the overall size of the client-side code. However, if the web application wants to enjoy the maintainability benefits of the UI framework, the JAVASCRIPT files should be imported anyway. As mentioned, this is an extensively-popular trend among the developers [242, 243]. Notwithstanding, if developers opt for using standard HTML Web Components [189] instead of third-party UI frameworks, there will be no burden in terms of the additional imported JAVASCRIPT files. As a result, when reporting the size measurements for REACT, we only consider the code generated by our approach for implementing UI components, not REACT's own core JAVASCRIPT code.

Results and Discussion

Figure 6.6 illustrates the results of applying the proposed refactorings on the test subjects. Observe that, using REACT implementation, refactoring UI components results in reducing the size of the HTML code by 6%–19.34%, with an average of 11.56%. The *intrinsic* performance of the algorithm itself, however, is higher: 14.90%–35.54%, with an average of 18.96%. This difference highlights that REACT components require quite considerable amount of added code to the original DOM information of the UI component instances. For example, a UI

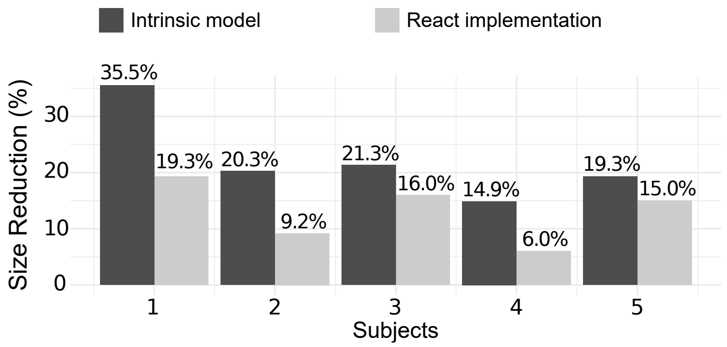


Figure 6.6: Code reusability achieved by the proposed component generation, as measured by final size reduction.

component shown in Figure 6.5(d) needs to be wrapped into a function named `render` implemented in a JAVASCRIPT class that extends the internal REACT class, `React.Component`. We also need to add additional code to pass arguments to the UI components for each UI component instance. As mentioned, using another UI framework can yield different saving ratios. It is for these reasons that reporting the intrinsic performance is important.

It is worth mentioning that this saving is not meant to replace existing techniques that, for instance, minify HTML by removing white space, or via any other reduction approach. Rather, whatever savings obtained from the components can complement them by adding even more saved bytes on top of what they would normally save.

6.5 Discussion

Context within web development. The approach we present in this chapter refactors repetitions in the UI and merges them into a template, which is finally converted into a component in one of the common front-end frameworks (e.g., RE-

ACT, ANGULAR). This automates one of the initial steps in making a full-fledged web app UI, which is often time consuming and is done manually. The use of this approach, of course, would not mean that the web app is ready to launch to clients and the development process is finished. The developer would use these components and continue the development, by, e.g., adding business logic, handling events, connecting to databases or other sources. Furthermore, the approach we present in this chapter is for modularizing the UI view itself, and is therefore orthogonal to the remaining components of the architecture pattern of the app (e.g., Model-view-controller (MVC), Model-view-presenter (MVP)) and to any backend server functionality.

Threats to validity. We chose test subjects (i.e., mockups) randomly from the Internet with the mentioned criteria in Section 6.4.2, to avoid any selection bias. Plus, the evaluation participants are expert web developers with different years of web development experience, mitigating the threats to the internal validity of the study. The mockups are diverse and complex enough to be representative of real-world app front-ends, mitigating the external validity of the study by making the results generalizable. To make the study replicable, we have made VIZMOD's source code, evaluation subjects, and the anonymized participants' responses available online [183].

6.6 Related Work

Visual analysis. There exist a few techniques that analyze web applications from a visual perspective. Choudhary et al. [65] propose an approach that detects cross-browser compatibility by examining visual differences between the same app running in multiple browsers. Burg et al. [52] present a tool that helps developers understand the behavior of front-end apps. It allows developers to specify which element they are interested in, then tracks that element for any visual changes and the corresponding code changes. Bajammal et al. [30] propose an approach to analyze and test web canvas element through visual inference of the state of the canvas and its objects, and allowing canvas elements to be testable using common DOM testing approaches. In contrast to our work, none of these studies aims to automatically identify and extract web components. Stocco et al. [153, 244] explore visual techniques for web testing applications, including visual-based test repair and techniques for migrating DOM-based tests to visual tests.

Clone detection. There is a large body of work on clone detection in conventional source code [208, 217, 219]. Some techniques also exist targeting web artifacts, such as for identifying duplicated content [44] or script function clones [55, 146], and quantifying the structural similarity across pages [77]. A number of existing publications [96, 159, 160, 174] propose template identification for Java code by defining a number of heuristics to compute code similarity. Rajapakse and Jarzabek [205] use CCFinder [133] to identify duplication in web applications. Synytskyy et al. [250] use an *island grammer* to identify cloned HTML forms and tables. Cordy et al. [71] propose a language-independent technique to identify exact/near-miss clones (initially in HTML) using island grammars, pretty-printing and textual differencing. Inspired by that work, NICAD clone detector is proposed [218].

Transformation and refactoring. Various techniques are proposed to convert static pages to dynamic ones [44, 250], to generalize dynamic web pages [76, 206], or to find similar functionalities across web pages [77]. Other techniques [180] use clustering to group similar static web pages together to extract single-page templates. Pattern mining techniques are used [176, 177, 178] for identifying and refactoring duplicated CSS code in web apps. In contrast to our work, none of these studies aims at automatically identifying and extracting web components from mockups.

6.7 Conclusions

The development of a web app front-end involves multiple stakeholders, chief among them the graphics designer and web developer. A UI mockup designed by the graphics designer has to be analyzed and processed by a web developer in order create the app's front-end code, a task that is laborious and involves manual time consuming steps. In this chapter, we proposed an approach to automate this aspect of web development by generating reusable web components from a mockup. We conducted an evaluation on real-world web mockups and assessed the quality of generated components through comparison with expert developers. An average of 94% precision and 75% recall was achieved in terms of agreement with the developers' assessment, the refactorings were performed in a correct manner, and the components achieved a 22% reusability, on average.

Chapter 7

Concluding Remarks

Building web UIs requires significant effort and time. Much of the state-of-the-art research on web UIs has typically revolved around analyzing the source code. While this approach has its uses and benefits, it is often not suitable for addressing non-functional properties, such as testability, accessibility, and maintainability. These aspects, while important, are generic and qualitative in nature compared to the more common functional properties or requirements which are precise and exact. Non-functional aspects therefore often require a high-level qualitative analysis which has made them less amenable to effective automation. Accordingly, analyzing or assessing such non-functional properties has often remained a laborious task that has to be done manually.

7.1 Contributions

In this dissertation, we introduced novel techniques for automating the analysis and testing of non-functional web UI properties (testability, accessibility, and maintainability) using visual approaches. Table 7.1 shows a summary of the contributions and the addressed research problems. The main contributions of this dissertation are as follows:

- An approach, in Chapter 3, that improves the testability of web UI by converting the non-testable canvas elements into testable ones. We presented a visual analysis technique that infers the objects, properties, and arrangements of the canvas element's content, then augments the inferred information into the DOM in order to make the canvas testable. The evaluation points to a high accuracy of inferring the contents of the canvas, and an effective detection of visual faults on the canvas.

- An automated technique that tests the semantic accessibility of web UIs, discussed in Chapter 4. The technique conducts a visual inference process to determine the semantic roles of various regions, then asserts that the UI's markup has explicitly expressed the visually inferred roles. The evaluation indicates high accuracy of detecting semantic groupings and inferring their semantic roles, as well as detecting both injected faults and real-world faults in test subjects.

- An approach for automated repair of inaccessible web form labeling, which is one of the most common causes for web inaccessibility. The approach, described in Chapter 5, is based on constructing a set of visual cues and then using them in formulating an optimization problem, the solution of which is applied to repair to the DOM. The evaluation shows that the DOM repair is conducted in a safe and effective manner, with a high accuracy of inferring labeling associations, and is sufficiently scalable for real-time usage.

Table 7.1: Summary of the targeted research problems and contributions.

Non-functional Property	Research Problems	Contributions
Accessibility	- testing semantic roles - repairing form labels	Existing approaches perform syntactic checks of accessibility. Semantic roles and form labels are not tested or repaired. The work in this dissertation tests roles and repairs labels by analyzing visual cues from the page.
Maintainability	- generating reusable components	Creating web UI components is a laborious manual effort. The work in this dissertation helps in automating the component creation process by analyzing visual patterns on the page.
Testability	- testing canvas elements	Canvas elements have not been amenable to testing due to absence of state. The work in this dissertation makes canvas elements testable by visually analyzing the canvas to infer its state.

- A technique that enhances the maintainability of web UI by automating some of the laborious steps of creating web components, which are one of the most common ways of creating maintainable UIs. The approach is based on analyzing abstract visual repetition trends, then combining them into reusable web templates, as described in Chapter 6. The evaluation points to an effective capture of repetitions in the UI, which achieved a high level of code reusability.

- A peer-reviewed systematic survey on the use of visual analysis in software engineering, which is the first to investigate this analysis paradigm. The goal of this work is to conduct a survey to help structure, curate, and unify the dispersed literature in this research area, and to analyze how visual techniques have been used in software engineering, and what are the challenges reported when they were used.

7.2 Research Questions Revisited

The research questions of this dissertation were introduced in Chapter 1. The questions were then investigated in the subsequent chapters (Chapter 2-6). We revisit each research question below and provide concluding remarks.

Research Question A

What areas of software engineering have benefited from visual analysis and how?

A recent, but scarcely explored, paradigm in software engineering research is to adopt a visual perspective of analyzing the software, which entails extracting and processing visual artifacts relevant to the software. To gain a better understanding of this trend, in this research question we surveyed the literature on the use of visual approaches in software engineering, as explored in Chapter 2. From an initial pool of 2,716 publications, we systematically obtained 66 papers and analyzed them according to a number of research dimensions (i.e., research area, technique, rationale). Our study revealed that visual techniques have been utilized most frequently in the software testing field. More specifically, most of the existing works utilizing visual analysis were in the research area of cross-browser testing, where the goal is to check whether or not a given app functions the same across browsers. These often use simple visual techniques such as screenshot differencing, which provides a simple measure of UI similarity between two browsers. Finally, our findings show that there has been little to no exploration of non-functional properties from a visual perspective, and therefore there is a research opportunity to explore such topics.

Research Question B

How can we make untestable web UI canvas elements testable through visual analysis?

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. In investigating this research question, we proposed a testing approach, discussed in Chapter 3, based on visual analysis of the screenshot of canvas elements in order to identify it structure and content, followed by 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 is able to reliably detect visual faults in canvases and infer its contents. These results indicate that, by visually deconstructing the content of the canvas and their properties, the canvas can be made testable.

Research Question C

How can we visually test the semantic accessibility of web UI?

In this research question, we introduced an approach that automates web accessibility testing from a semantic perspective, as described in Chapter 4. While some tools exist to perform basic forms of accessibility checks, they focus on syntactic checks, as opposed to checking the more critical high level semantic accessibility features that users with disabilities rely on. The proposed approach in this research question analyzes web pages using a combination of visual cues, and infers the semantic groupings present in the page and their semantic roles. It then asserts whether the page's markup matches the inferred semantics. We evaluated our approach on real-world websites and assessed the accuracy of semantic inference as well as its ability to detect accessibility failures. The results show, on average, an F-measure of 87% for inferring semantic groupings, and an accessibility failures detection accuracy of 85%.

Research Question D

Can we automatically repair the accessibility of web UI forms through visual analysis?

Filling web forms is a key activity while browsing the web. While this task can be easily completed by sighted users, it is a significant hurdle for non-sighted users if the form does not contain the required accessibility labeling. This issue of missing form labeling is consistently one of the top three most common web accessibility issues. Unfortunately, however, when a non-sighted user is faced with a non-accessible form, there are currently little to no options available to access that form. To this end, in this research question, we introduced an approach that automatically analyzes web forms and makes them accessible. The approach first abstracts a given web form, then generates visual cues from the form, in an effort to emulate how sighted users would visually perceive the form. The visual cues are then used in a constrained binary optimization program to solve for the form labeling associations. These are finally translated into standard ARIA accessibility markups and augmented into the DOM to repair the form and make it accessible. We evaluated our approach on real-world subjects and assessed the accuracy of labeling inference, the safety of the DOM augmentation repairs, as well as the labeling performance. The results show an average F1-measure of 88.1% for label inference, and an average run-time of around 1240 milliseconds.

Research Question E

How can we generate reusable web UI components through visual analysis?

The development of a web app front-end involves multiple stakeholders, chief among them the graphics designer and web developer. A UI mockup designed by the graphics designer has to be analyzed and processed by a web developer in order create the app's front-end code, a task that is laborious and involves manual time consuming steps. In this research question, we introduced an approach to automate this aspect of web development by generating reusable web components from a mockup. The approach is based on detecting visual patterns on the mockup, then combining them into components. The evaluation on real-world web subjects indicates that an average F1-score of 77.4% in terms of agreement with the developers' manual selection, performs the refactorings in a correct manner, and the components achieve a 22% code reusability, on average.

7.3 Reflections and Future Directions

In this dissertation, we took a first step towards using visual analysis for improving non-functional web UI properties. More specifically, the techniques we introduced have converted untestable elements into testable ones, tested the semantic accessibility of web UI, augmented inaccessible web forms into accessible ones, and automatically generated reusable web UI components. However, there still remains many avenues for future work.

7.3.1 Other non-functional aspects

Usability. Our research has shown that visual analysis is effective for testing accessibility issues. There are other non-functional aspects, however, that would likely benefit from a similar qualitative and high-level visual analysis. A potential research direction is to therefore investigate the use of visual analysis for other important non-functional properties such usability. Usability has become a key differentiating factor between different software products, with significant industrial interest in perfecting the user experience. This resulted in a growing demand for techniques that can assist with this aspect of development. Our research has shown that visual analysis is effective for accessibility issues, which are similar to usability in the sense that they require a qualitative high-level analysis. Accordingly, we foresee that usability issues can be captured and measured using visual analysis. For instance, an example of usability guideline is providing users with a response to each action. This guideline could be tested, for instance, by asserting the presence of visual updates or indicators after clicking on various UI objects in an app. If there is a lack of visual updates or indicators after the action, then this would be flagged as usability issue and reported to developers.

Performance There is also the potential to explore the performance of UI from a visual analysis perspective. Instead of separately measuring each component (e.g., network lag, backend processing), measuring it from a visual perspective would give the most realistic measure of performance. This might involve, for instance, detecting visual screen transitions and recording relevant durations. Such an approach would therefore mimic how end users perceive the performance of a UI, which would potentially yield more accurate measures of any lag or performance degradation experienced by the end user.

Accessibility platform. Another possible research direction is to create a modular framework that can adapt to various accessibility rules. While our research took a first step in the direction of automating some of the more pressing accessibility issues, there is definitely room to explore the other accessibility rules and guidelines. Current, legally accepted, accessibility guidelines often contain hundreds of rules at varying levels of granularity and abstraction. We foresee the potential to create a modular engine that allows different accessibility rules to be expressed in a standard and generic format. This can help, for instance, in building a developer community around a common analysis engine and therefore help automate a greater portion of accessibility requirements. The research challenge, however, is to design a framework that is flexible and adaptable enough to be able to capture a wide variety of accessibility rules.

Alternative maintainability approaches. Another potential avenue is to explore other approaches to improve maintainability. Our research has shown that visual analysis can effectively detect patterns in a UI and create reusable components to improve maintainability. We believe there is a potential to explore other aspects of maintainability in the same manner. For instance, we foresee that it can be useful to visually analyze UIs with the goal of removing or trimming down unnecessary markups and styles. This might be achieved in an automated fashion through iterative visual analysis by measuring, for instance, the impact of removing various pieces of markup and styles in order to ensure that they do not impact the UI visually. Another option would be to even predict that impact through a machine learning model that uses visual features, and then using the model as a tool during development and testing. These UI trimming approaches would have the benefit of reducing the execution runtime, the network traffic, and the size of the code base in order to make the UI code smaller and easier to maintain.

7.3.2 Other topics

Alternative platforms. In this dissertation, we addressed research problems for web applications. However, we foresee a potential for future work in other emerging platforms, such as virtual reality. For more traditional platforms, such as web, mobile, and desktop, visual analysis is relatively similar since they are all based on having common GUIs and elements. While the underlying technology implementations for these platforms might be different (e.g., XML instead of HTML, SVG instead of Canvas), they are still made of GUI elements, and therefore would have a similar visual analysis approach. For instance, the canvas analysis work that we have conducted can also be used with SVG elements, but it would not be needed in most cases since SVG elements already have a state representation and can be directly tested. The visual analysis process itself, however, is still applicable, and therefore can be used if the need arises for it. On the other hand, the techniques in this dissertation can not readily address the emerging virtual reality applications due to a number research challenges. These applications often have very rich visual interactions and details. Analyzing and testing them is more complex compared to web apps due to a number of factors, such as having new modalities of display (e.g., head-mounted displays) and input (e.g., spatial controllers), as well as the additional third dimension and the higher level of visual complexity. Analyzing and testing such applications would therefore require being able to handle the resulting explosion of the event space (e.g., combination of controller orientations and gestures) and state space (e.g., various angles and views), as well as having features that can capture the increased visual complexity (e.g., additional dimensions, complex free-form objects). These can then be tracked across temporal sequences of states in order to, for instance, test or analyze movements in the virtual space.

Deep learning. We expect that there will be more techniques that use deep learning to analyze visual aspects of software. This is because as the software analysis task being addressed gets more complex (e.g., harder to automate), it would likely require collecting and analyzing a large set of features that may not be easily determined from the outset, and therefore using a deep learning approach would reduce the effort required for feature selection. In our work, we have used a convolutional neural network in order to help with determining the accessibility roles of regions on the page. Many more venues for future work are possible. For instance, one possible line of work might involve building learning models for recognizing basic UI aspects such menus. This information about the presence of a menu and its individual items can then be used for various analysis or testing tasks, such as devising app exploration strategies that take into account the menu structure and content. Another potential research avenue is to learn models that can classify or categorize different types of user interfaces (e.g., login, settings). These can then be used to optimize the testing of an application by learning which pattern of test inputs are associated with what type of user interface, or can be also used in state abstraction and model building. These are certainly only a couple of examples, and the potential areas of applying deep learning are practically unlimited.

Games, animations, and complex visuals. Web applications are typically composed of GUI elements (e.g., buttons, menus), whereas other applications such as games and animations are much more visually rich and complex. The analysis we conducted in this dissertation was limited to web applications, and therefore the visual features (e.g., colors, shapes, locations) were two dimensional and static in nature. This is a limitation of our approach, however, when working with games and animations. Such applications would require extrapolating the features to three dimensions and shapes and objects of generic complexity, and then subsequently tracking and recording each feature across a temporal series of states in order to analyze the animation. However, we do point out that most games and animations do not perform the rendering on their own, but rather rely on common, industry-standard, rendering engines (e.g., Unity). Accordingly, visual analysis and testing would likely be of interest to the developers of such engines to a greater degree than developers of games and animations. This is because engine developers are the ones who are concerned with making sure that each particular rendering function correctly performs the rendering as visually expected. While developers of games and animations do also need some form of visual testing, they will likely focus on more higher-level aspects, such as asserting particular states or game logic elements, more than the low-level rendering accuracy. Even such high-level testing would likely have its own challenges, since the event and state space of games and animations is much larger than typical web applications. Handling the large state space would likely require researching appropriate abstraction strategies in order to simplify the state space and prune unnecessary details, and such abstraction will likely be application and task-specific. Furthermore, the goal of the canvas testability work in this dissertation is to provide developers with the fundamental capability of observing the canvas state and making assertions on it. However, it does not provide a complete testing solution, but rather make it *possible* to perform the testing process itself, thereby improving testability. As part of future work, a more through canvas testing solution can be provided such that it will cover more complex and resizable canvas elements, and generate the tests in a fashion that developers might prefer (e.g., using relative positioning in assertions).

Functional aspects. The work in this dissertation has focused on particular problems in non-functional analysis because they have not been amenable to automation and have received little attention so far. On the other hand, functional aspects have been more thoroughly researched and the earliest usage of visual analysis was to address functional aspects, such as regression testing of web UIs. We therefore expect functional aspects to continue being the most common application for visual analysis. However, we expect that potential future work would likely involve the use of more fine-grained visual analysis instead of direct image diffing, which is currently the most commonly used technique. This is because the fine-grained analysis would operate at the level of finer features of the user interface instead of the image as a whole, and therefore would potentially provide more accurate and robust results when conducting, for instance, regression testing or specifying oracles. In general, however, the nature of the functional testing would remain the same, in the sense that inputs are generated and certain aspects of the user interface are asserted. Nonetheless, as explored and demonstrated in this dissertation, it is the non-functional properties that have been least explored and therefore would benefit from new techniques that could advance state-of-the-art.

Bibliography

[1] World wide web consortium. accessible rich internet applications 1.2. 2019. → pages 3, 112, 144, 161

[2] Alexa top sites. https://aws.amazon.com/alexa-top-sites/. → page 165

[3] Iso/iec 25010:2011 systems and software engineering. → page 7

[4] Lpsolve mixed-integer linear programming system. https://cran.r-project.org/web/packages/lpSolve/index.html. → pages 164, 173

[5] W3C canvas elements standard. URL http://www.w3.org/TR/2dcontext/. → pages 5, 75

[6] J. Abascal, M. Arrue, and X. Valencia. Tools for web accessibility evaluation. In *Web Accessibility*, pages 479–503. Springer, 2019. → page 175

[7] S. Abou-Zahra. Web accessibility evaluation. In *Web accessibility*, pages 79–106. Springer, 2008. → pages 107, 145

[8] P. Acosta-Vargas, S. Luján-Mora, T. Acosta, and L. Salvador-Ullauri. Toward a combined method for evaluation of web accessibility. In *International Conference on Information Theoretic Security*, pages 602–613. Springer, 2018. → pages 3, 107, 145

[9] P. Acosta-Vargas, S. Luján-Mora, T. Acosta, and L. Salvador-Ullauri. Toward a combined method for evaluation of web accessibility. In *International Conference on Information Theoretic Security*, pages 602–613. Springer, 2018. → page 6

[10] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce. Combinatorial-based event sequence testing of android applications. *Information and Software Technology*, 99:98–117, 2018. → page 176

[11] G. Agrawal, D. Kumar, M. Singh, and D. Dani. Evaluating accessibility and usability of airline websites. In *International Conference on Advances in Computing and Data Sciences*, pages 392–402. Springer, 2019. → pages 141, 176

[12] E. Akpınar and Y. Yeşilada. " old habits die hard!" eyetracking based experiential transcoding: a study with mobile users. In *Proceedings of the 12th International Web for All Conference*, pages 1–5, 2015. → page 175

[13] A. Al-Kaff, D. Martín, F. García, A. de la Escalera, and J. María Armingol. Survey of computer vision algorithms and applications for unmanned aerial vehicles. *Expert Systems with Applications*, 92:447–463, 2018. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2017.09.033. URL http://www.sciencedirect.com/science/article/pii/S0957417417306395. → page 12

[14] E. Alégroth and R. Feldt. On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, 22(6): 2937–2971, 2017. → page 15

[15] E. Alégroth, M. Nass, and H. H. Olsson. JAutomate: A Tool for System- and Acceptance-test Automation. In *Proc. of ICST '13*, pages 439–446, 2013. → pages 10, 29, 34, 40, 43, 46, 48, 49, 60, 61, 68

[16] E. Alégroth, Z. Gao, R. Oliveira, and A. Memon. Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015. → page 15

[17] E. Alégroth, A. Karlsson, and A. Radway. Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 172–181, 2018. → page 15

[18] A. Alshayban, I. Ahmed, and S. Malek. Accessibility issues in android apps: state of affairs, sentiments, and ways forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1323–1334. IEEE, 2020. → pages 141, 145, 176

[19] D. Amalfitano, A. R. Fasolino, S. Scala, and P. Tramontana. Towards Automatic Model-in-the-loop Testing of Electronic Vehicle Information

Centers. In *Proc. of WISE '14*, pages 9–12, 2014. → pages 29, 34, 39, 40, 41, 46, 48, 49, 53, 60, 61

[20] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh. Mobolic: An automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation. *Software: Practice and Experience*, 48(5):1107–1142, 2018. → page 177

[21] R. Arreola. *Creating visuals for math instruction using JavaScript and the HTML canvas element*. PhD thesis, California State University, Northridge, 2017. → page 71

[22] C. Asakawa, H. Takagi, and K. Fukuda. Transcoding. In *Web Accessibility*, pages 569–602. Springer, 2019. → pages 145, 174

[23] ASE 2018 repository: . https://github.com/msbajammal/vizmod. → page vi

[24] ASLint. Accessibility linter., 2020. URL https://www.aslint.org/. → pages 141, 176

[25] A. Bai, H. C. Mork, T. Schulz, and K. S. Fuglerud. Evaluation of accessibility testing methods. which methods uncover what type of problems? *Studies in health technology and informatics*, 229:506–516, 2016. → pages 3, 107, 145

[26] A. Bai, H. C. Mork, T. Schulz, and K. S. Fuglerud. Evaluation of accessibility testing methods. which methods uncover what type of problems? *Studies in health technology and informatics*, 229:506–516, 2016. → page 6

[27] A. Bai, K. S. Fuglerud, R. A. Skjerve, and T. Halbach. Categorization and comparison of accessibility testing methods for software development. 2018. → pages 141, 176

[28] A. Bai, V. Stray, and H. Mork. What methods software teams prefer when testing web accessibility. *Advances in Human-Computer Interaction*, 2019, 2019. → pages 141, 176

[29] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 112–121. ACM, 2014. → page 102

[30] M. Bajammal and A. Mesbah. Web canvas testing through visual inference. In *11th IEEE International Conference on Software Testing, Verification*

and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018, pages 193–203. IEEE Computer Society, 2018. → pages v, 29, 33, 37, 40, 46, 48, 50, 53, 59, 61, 63, 212

[31] M. Bajammal and A. Mesbah. Semantic web accessibility testing via hierarchical visual analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1610–1621. IEEE, 2021. → pages vi, 3

[32] M. Bajammal, D. Mazinanian, and A. Mesbah. Generating reusable web components from mockups. In M. Huchard, C. Kästner, and G. Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 601–611. ACM, 2018. → pages vi, 29, 38, 40, 48, 61, 142, 177

[33] M. Bajammal, A. Stocco, D. Mazinanian, and A. Mesbah. A survey on the use of computer vision to improve software engineering tasks. *IEEE Transactions on Software Engineering (TSE)*, 2020. → pages v, 2

[34] L. Bao, J. Li, Z. Xing, X. Wang, and B. Zhou. scvripper: video scraping tool for modeling developers' behavior using interaction data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 673–676. IEEE, 2015. → pages 29, 38, 40, 48, 55, 61

[35] L. Bao, J. Li, Z. Xing, X. Wang, X. Xia, and B. Zhou. Extracting and Analyzing Time-series HCI Data from Screen-captured Task Videos. *ESEM*, 22(1):134–174, 2017. → pages 29, 40, 47, 48, 50, 52, 53, 54, 55, 59, 61, 62, 64

[36] L. Bao, Z. Xing, X. Xia, and D. Lo. Vt-revolution: Interactive programming video tutorial authoring and watching system. *IEEE Transactions on Software Engineering*, 2018. → pages 29, 40, 48, 53, 55, 61

[37] J. A. Bargas-Avila, O. Brenzikofer, S. Roth, A. Tuch, S. Orsini, and K. Opwis. Simple but crucial user interfaces in the world wide web: introducing 20 guidelines for usable web form design, user interfaces. 2010. → page 153

[38] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006. → page 54

[39] N. L. Bayer and L. Pappas. Accessibility testing: Case history of blind testers of enterprise software. *Technical Communication*, 53(1):32–38, 2006. → pages 107, 145

[40] W. Behutiye, P. Karhapää, D. Costal, M. Oivo, and X. Franch. Non-functional requirements documentation in agile software development: challenges and solution proposal. In *International conference on product-focused software process improvement*, pages 515–522. Springer, 2017. → pages 1, 3

[41] Á. Beszédes. Interdisciplinary survey of fault localization techniques to aid software engineering. *Acta Polytechnica Hungarica*, 16:207–226, 2019. → page 13

[42] S. Bhagat and P. Joshi. Evaluation of accessibility and accessibility audit methods for e-governance portals. In *Proceedings of the 12th International Conference on Theory and Practice of Electronic Governance*, pages 220–226, 2019. → pages 141, 145, 176

[43] J. P. Bigham, R. S. Kaminsky, R. E. Ladner, O. M. Danielsson, and G. L. Hempton. Webinsight: making web images accessible. In *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 181–188, 2006. → page 175

[44] C. Boldyreff and R. Kewish. Reverse engineering to achieve maintainable WWW sites. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 249–257, 2001. ISBN 0-7695-1303-4. → page 212

[45] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. → pages 128, 164

[46] H. Braga, L. S. Pereira, S. B. L. Ferreira, and D. S. Da Silveira. Applying the barrier walkthrough method: Going beyond the automatic evaluation of accessibility. *Procedia Computer Science*, 27:471–480, 2014. → pages 107, 145

[47] G. Brajnik. A comparative test of web accessibility evaluation methods. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*, pages 113–120. ACM, 2008. → pages 107, 145

[48] R. J. Breeds and J. S. Taylor. Software accessibility testing, Mar. 4 2014. US Patent 8,667,468. → pages 107, 145

[49] A. Brunetti, D. Buongiorno, G. F. Trotta, and V. Bevilacqua. Computer vision and deep learning techniques for pedestrian detection and tracking: A survey. *Neurocomputing*, 300:17–33, 2018. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2018.01.092. URL http://www.sciencedirect.com/science/article/pii/S092523121830290X. → page 12

[50] 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*, 17(1):66, 2016. → page 71

[51] B. Burg, A. J. Ko, and M. D. Ernst. Explaining Visual Changes in Web Interfaces. In *Proc. of UIST '15*, pages 259–268, 2015. → pages 29, 37, 40, 46, 48, 49, 52, 60, 61

[52] 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*, pages 259–268. ACM, 2015. → pages 104, 142, 177, 211

[53] G. Buscher, E. Cutrell, and M. R. Morris. What do you see when you're surfing? using eye tracking to predict salient regions of web pages. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 21–30, 2009. → page 152

[54] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge. Javasketchit: Issues in sketching the look of user interfaces. In *AAAI Spring Symposium on Sketch Understanding*, pages 9–14, 2002. → pages 29, 36, 40, 48, 61

[55] F. Calefato, F. Lanubile, and T. Mallardo. Function clone detection in web applications: a semiautomated approach. *Journal of Web Engineering*, 3 (1):3–21, 2004. → page 212

[56] R. J. Campello, D. Moulavi, and J. Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer, 2013. → pages 194, 200

[57] CanvaSure. repository. https://github.com/msbajammal/canvasure. → page 93

[58] P. Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics*, 17(7): 913–933, 2010. → page 14

[59] Centers for Disease Control and Prevention, National Center on Birth Defects and Developmental Disabilities, Division of Human Development and Disability. Disability and health data system (dhds) data. https://dhds.cdc.gov, 2016. → pages xv, 6, 108, 144

[60] T.-H. Chang, T. Yeh, and R. C. Miller. GUI Testing Using Computer Vision. In *Proc. of CHI '10*, pages 1535–1544, 2010. → pages 10, 29, 33, 34, 40, 43, 46, 48, 50, 53, 55, 61, 63, 64

[61] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 665–676, 2018. → pages 29, 38, 40, 48, 61

[62] C.-F. R. Chen, M. Pistoia, C. Shi, P. Girolami, J. W. Ligman, and Y. Wang. UI X-Ray: Interactive Mobile UI Testing Based on Computer Vision. In *Proc. of IUI '17*, pages 245–255, 2017. → pages 29, 40, 41, 46, 48, 50, 52, 53, 59, 61

[63] 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*, pages 1–10. IEEE, 2010. → page 104

[64] S. R. Choudhary, M. R. Prasad, and A. Orso. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proc. of ICST '12*, pages 171–180, 2012. → pages 29, 34, 40, 41, 42, 46, 48, 49, 52, 54, 55, 59, 60, 61

[65] S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 171–180. IEEE, 2012. → pages 142, 177, 211

[66] S. R. S. Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proc. of*

ICSM '10, pages 1–10, 2010. → pages 29, 34, 40, 41, 46, 48, 49, 52, 55, 59, 60, 61, 63, 64, 66

[67] 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)*, 1(3):16–25, 2012. → page 71

[68] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012. → page 2

[69] T. W. W. W. Consortium. Using grouping roles to identify related form controls. https://www.w3.org/TR/WCAG20-TECHS/ARIA17.html, 2021. → page 163

[70] 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*, INTUITEST 2016, pages 11–20, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4412-8. doi: 10.1145/2945404.2945406. URL http://doi.acm.org/10.1145/2945404.2945406. → pages 73, 103

[71] J. R. Cordy and T. R. Dean. Practical language-independent detection of near-miss clones. In *Proceedings of the 14th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 1–12, 2004. → page 212

[72] N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–114, 2001. → page 207

[73] A. Coyette, S. Kieffer, and J. Vanderdonckt. Multi-fidelity prototyping of user interfaces. In *IFIP Conference on Human-Computer Interaction*, pages 150–164. Springer, 2007. → pages 29, 36, 40, 48, 61

[74] 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*, 42(D1):D472, 2014. doi: 10.1093/nar/gkt1102. URL +http://dx.doi.org/10.1093/nar/gkt1102. → pages 95, 101

[75] I. Daubechies. The wavelet transform, time-frequency localization and signal analysis. *IEEE transactions on information theory*, 36(5):961–1005, 1990. → page 54

[76] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Reengineering web applications based on cloned pattern analysis. In *Proceedings of 12th IEEE International Workshop on Program Comprehension*, pages 132–141. IEEE, 2004. ISBN 0-7695-2149-5. → page 212

[77] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Understanding cloned patterns in web applications. In *Proceedings of the 13th International Workshop on Program Comprehension (ICPC)*, pages 333–336. IEEE, 2005. ISBN 0-7695-2254-8. → page 212

[78] B. Deka, Z. Huang, and R. Kumar. ERICA: Interaction Mining Mobile Apps. In *Proc. of UIST '16*, pages 767–776, 2016. → pages 29, 35, 36, 37, 38, 40, 46, 48, 50, 60, 61

[79] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proc. of UIST '17*, pages 845–854, 2017. → pages 29, 35, 36, 37, 38, 40, 46, 48, 49, 52, 61, 68

[80] M. E. Delamaro, L. dos Santos Nunes Fátima, and O. R. A. Paes. Using concepts of content-based image retrieval to implement graphical testing oracles. *STVR*, 23(3):171–198, 2011. → pages 29, 40, 41, 46, 48, 49, 59, 61, 68

[81] M. Dhok, M. K. Ramanathan, and N. Sinha. Type-aware concolic testing of javascript programs. In *Proceedings of the 38th International Conference on Software Engineering*, pages 168–179, 2016. → page 177

[82] M. Dixon and J. Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1525–1534, 2010. → pages 29, 40, 48

[83] M. Dixon, D. Leventhal, and J. Fogarty. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 969–978. ACM, 2011. → pages 29, 37, 40, 48

[84] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *EMSE*, 10(4):405–435, 2005. → page 67

[85] T. Domínguez Vila, E. Alén González, and S. Darcy. Website accessibility in the tourism industry: an analysis of official national tourism organization websites around the world. *Disability and rehabilitation*, 40(24): 2895–2906, 2018. → pages 141, 176

[86] D. A. Dondis. *A primer of visual literacy*. MIT Press, 1974. → page 183

[87] C. E. Ebeling. *An Introduction to Reliability and Maintainability Engineering*. Waveland Press, 2019. → page 7

[88] Eclipse Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0. URL http://deeplearning4j.org. → page 128

[89] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126. IEEE, 2018. → pages 107, 141, 145, 176

[90] S. Eraslan, Y. Yesilada, and S. Harper. Eye tracking scanpath analysis techniques on web pages: A survey, evaluation and comparison. *Journal of Eye Movement Research*, 9(1), 2016. → page 152

[91] European Union's Eurostat. European union labour force survey (eu-lfs). prevalence of disability. http://appsso.eurostat.ec.europa.eu, 2014. → pages xv, 108, 144

[92] J. Fails and D. Olsen. A design tool for camera-based interaction. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 449–456, 2003. → pages 29, 38, 40, 48, 61

[93] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(4):594–611, Apr. 2006. ISSN 0162-8828. doi: 10.1109/TPAMI.2006.79. URL https://doi.org/10.1109/TPAMI.2006.79. → page 67

[94] Y. Feng, J. A. Jones, Z. Chen, and C. Fang. Multi-objective Test Report Prioritization Using Image Understanding. In *Proc. of ASE '16*, pages 202–213, 2016. → pages 29, 35, 40, 46, 48, 49, 52, 55, 59, 61

[95] C. Fernandez-Maloigne. *Advanced color image processing and analysis*. Springer Science & Business Media, 2012. → page 80

[96] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 391–400. IEEE, 2014. → page 212

[97] R. Z. Frantz, M. H. Rehbein, R. Berlezi, and F. Roos-Frantz. Ranking open source application integration frameworks based on maintainability metrics: A review of five-year evolution. *Software: Practice and Experience*, 49(10):1531–1549, 2019. → page 8

[98] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proc. of ICSE '12*, pages 178–188, 2012. → page 67

[99] G. Gan, C. Ma, and J. Wu. *Data clustering: Theory, algorithms, and applications*. Society for Industrial and Applied Mathematics, 2021. → page 160

[100] T. Gandhi and M. M. Trivedi. Pedestrian collision avoidance systems: a survey of computer vision based recent studies. In *2006 IEEE Intelligent Transportation Systems Conference*, pages 976–981, 2006. → page 12

[101] V. Garousi, W. Afzal, A. Çağlar, İ. B. Işık, B. Baydan, S. Çaylak, A. Z. Boyraz, B. Yolaçan, and K. Herkiloğlu. Comparing automated visual gui testing tools: an industrial case study. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, pages 21–28. ACM, 2017. → page 15

[102] V. Garousi, M. Felderer, and F. N. Kılıçaslan. A survey on software testability. *Information and Software Technology*, 108:35–64, 2019. → page 5

[103] P. Givens, A. Chakarov, S. Sankaranarayanan, and T. Yeh. Exploring the internal state of user interfaces by combining computer vision techniques with grammatical inference. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1165–1168. IEEE, 2013. → pages 29, 37, 40, 48, 55, 61

[104] Google Inc. Angular, 2016. URL https://angular.io/. Accessed: 15 February 2018. → page 180

[105] Google Inc. Angular core documentation: Components, 2017. URL https://angular.io/api/core/Component. Accessed: 4 April 2018. → page 180

[106] Google Inc. Angular core documentation: Components, 2018. URL https://angular.io/api/core/Component. Accessed: 4 April 2018. → page 8

[107] Google Inc. Accessibility scanner., 2020. URL https://support.google.com/accessibility/android/answer/6376570. → pages 141, 176

[108] P. Gupta, S. Gupta, A. Jayagopal, S. Pal, and R. Sinha. Saliency prediction for mobile user interfaces. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1529–1538. IEEE, 2018. → page 152

[109] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah. BugsJS: a benchmark of JavaScript bugs. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, page 12 pages. IEEE Computer Society, 2019. URL /publications/docs/icst19.pdf. → page 67

[110] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Árpád Beszédes, R. Ferenc, and A. Mesbah. BugJS: A benchmark and taxonomy of javascript bugs. *Software Testing, Verification And Reliability*, 2020. → page 67

[111] M. Hadjieleftheriou, Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras. R-trees–a dynamic index structure for spatial searching. In *Encyclopedia of GIS*, pages 993–1002. Springer, 2008. → page 83

[112] M. Hammoudi, G. Rothermel, and A. Stocco. WATERFALL: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 751–762. ACM, 2016. ISBN 978-1-4503-4218-6. → page 43

[113] M. Hammoudi, G. Rothermel, and P. Tonella. Why do record/replay tests of web applications break? In *Proc. of ICST '16*, pages 180–190, 2016. → page 43

[114] S. Harper and A. Q. Chen. Web accessibility guidelines. *World Wide Web*, 15(1):61–88, 2012. → pages 106, 145

[115] M. He, G. Wu, H. Tang, W. Chen, J. Wei, H. Zhong, and T. Huang. X-Check: A Novel Cross-Browser Testing Service Based on Record/Replay. In *Proc. of ICWS '16*, pages 123–130, 2016. → pages 29, 34, 40, 41, 46, 48, 49, 52, 59, 61, 66

[116] P. Hegedűs, I. Kádár, R. Ferenc, and T. Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95:313–327, 2018. → page 8

[117] A. Hori, S. Takada, H. Tanno, and M. Oinuma. An Oracle based on Image Comparison for Regression Testing of Web Applications. In *Proc. of SEKE '15*, SEKE '15, pages 639–645, 2015. → pages 29, 40, 46, 48, 49, 52, 59, 60, 61

[118] M. Z. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CsUR)*, 51(6):1–36, 2019. → page 175

[119] F. Huang, J. F. Canny, and J. Nichols. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 104. ACM, 2019. → pages 29, 40, 48, 61

[120] ICSE 2021 repository: . https://github.com/msbajammal/icse2021. → page vi

[121] IDC. Mobile trends report, 2015. https://www.appcelerator.com/resource-center/research/2015-mobile-trends-report/. Accessed: 15 February 2018. → pages 1, 179

[122] Institute for Disability Research, Policy, and Practice. Utah State University. Annual accessibility analysis. https://webaim.org/projects/million/, 2021. → page 7

[123] Institute for Disability Research, Policy, and Practice. Utah State University. Annual accessibility analysis. https://webaim.org/projects/million/, 2021. → pages 144, 145, 175

[124] 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)*, pages 11–15, Sept. 2012. doi: 10.1109/WSE.2012.6320526. → page 15

[125] J. Janai, F. Güney, A. Behl, and A. Geiger. Computer vision for autonomous vehicles: Problems, datasets and state-of-the-art. *CoRR*, abs/1704.05519, 2017. URL http://arxiv.org/abs/1704.05519. → page 12

[126] N. Jha and A. Mahmoud. Mining non-functional requirements from app store reviews. *Empirical Software Engineering*, 24(6):3659–3695, 2019. → page 3

[127] E. Jones, T. Oliphant, and P. Peterson. {SciPy}: open source scientific tools for {Python}. 2014. → page 93

[128] Jordan Walke, Facebook, Instagram and community. React - A JavaScript library for building user interfaces, 2013. URL https://reactjs.org/. Accessed: 15 February 2018. → pages 180, 195

[129] Jordan Walke, Facebook, Instagram, and community. React documentation: React.component, 2013. URL https://reactjs.org/docs/react-component.html. Accessed: 4 April 2018. → page 180

[130] Jordan Walke, Facebook, Instagram and community. JSX Whitespace, 2014. URL https://reactjs.org/blog/2014/02/20/react-v0.9.html#jsx-whitespace. Accessed: 16 April 2018. → page 202

[131] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 485–495, 2009. ISBN 978-1-4244-3453-4. → page 182

[132] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. of the ISSTA '14*, pages 437–440, 2014. → page 67

[133] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, 2002. ISSN 0098-5589. → page 212

[134] C. Kanellakis and G. Nikolakopoulos. Survey on computer vision for uavs: Current developments and trends. *Journal of Intelligent & Robotic Systems*, 87:141–168, 2017. → page 11

[135] R. Kimmons. Open to all? nationwide evaluation of high-priority web accessibility considerations among higher education websites. *Journal of Computing in Higher Education*, 29(3):434–450, 2017. → page 141

[136] H. Kirinuki, H. Tanno, and K. Natsukawa. Color: correct locator recommender for broken test scripts using various clues in web application. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 310–320. IEEE, 2019. → page 172

[137] N. Kiryati, H. Kälviäinen, and S. Alaoutinen. Randomized or probabilistic hough transform: unified performance evaluation. *Pattern Recognition Letters*, 21(13):1157–1164, 2000. → page 83

[138] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. 2007. → page 16

[139] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003. → page 15

[140] M. F. Kıraç, B. Aktemur, and H. Sözer. VISOR: A fast image processing pipeline with scaling and translation invariance for test oracle automation of visual output systems. *JSS*, 136:266–277, 2018. → pages 29, 33, 39, 40, 46, 48, 49, 52, 53, 59, 61, 63, 64

[141] E. Krainz, K. Miesenberger, and J. Feiner. Can we improve app accessibility with advanced development methods? In *International Conference on Computers Helping People with Special Needs*, pages 64–70. Springer, 2018. → pages 141, 176

[142] D. Kroon. Numerical optimization of kernel based image derivatives. *Short Paper University Twente*, 2009. → page 80

[143] T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in pdf readers and files. *EMSE*, 2018. → pages 29, 32, 34, 40, 41, 46, 48, 50, 54, 59, 60, 61, 62, 63

[144] A. Kumar. Computer-vision-based fabric defect detection: A survey. *IEEE Transactions on Industrial Electronics*, 55(1):348–363, 2008. → page 11

[145] J. A. Landay and B. A. Myers. Sketching interfaces: Toward more human interface design. *Computer*, 34(3):56–64, 2001. → pages 29, 36, 40, 48

[146] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 379–386, 2003. ISBN 0-7695-1902-4. → page 212

[147] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-Based Web Locators: An Empirical Study. In *Web Engineering*, pages 322–340. Springer, Cham, July 2014. URL http://link.springer.com/chapter/10.1007/978-3-319-08245-5_19. DOI: 10.1007/978-3-319-08245-5_19. → pages 73, 103

[148] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation*, ICST '15, pages 1–10. IEEE, 2015. → page 43

[149] 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*, pages 775–782. ACM, 2015. → page 104

[150] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, pages 28:177–204, 2016. → page 43

[151] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *JSEP*, 28(3):177–204, 2016. ISSN 2047-7481. → page 43

[152] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016. → page 172

[153] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. PESTO: Automated migration of DOM-based web tests towards the visual approach. *Software Testing, Verification And Reliability*, 28(4):e:1665, 2018. → page 212

[154] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. PESTO: Automated migration of DOM-based web tests towards the visual approach. *STVR*, 28 (4), 2018. → pages 29, 32, 37, 40, 46, 48, 50, 53, 55, 59, 61, 62, 63, 64

[155] Y. Li, X. Cao, K. Everitt, M. Dixon, and J. A. Landay. FrameWire: A Tool for Automatically Extracting Interaction Logic from Paper Prototyping Tests. In *Proc. of CHI '10*, pages 503–512, 2010. → pages 10, 29, 35, 37, 40, 47, 48, 50, 52, 53, 59, 60, 61, 62, 63

[156] H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen. Seess: seeing what i broke–visualizing change impact of cascading style sheets (css). In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 353–356, 2013. → pages 29, 40, 48, 61

[157] W. Lidwell, K. Holden, and J. Butler. *Universal principles of design, revised and updated*. Rockport Pub, 2010. → page 183

[158] S. Lim, J. Hibschman, H. Zhang, and E. O'Rourke. Ply: A visual web inspector for learning from professional webpages. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 991–1002, 2018. → pages 29, 37, 40, 48, 61

[159] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 520–531. ACM, 2015. → page 212

[160] Y. Lin, G. Meng, Y. Xue, Z. Xing, J. Sun, X. Peng, Y. Liu, W. Zhao, and J. Dong. Mining implicit design templates for actionable code reuse. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 394–404. IEEE, 2017. → page 212

[161] Y. D. Lin, J. F. Rojas, E. T. H. Chu, and Y. C. Lai. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *TSE*, 40(10):957–970, 2014. → pages 10, 29, 34, 40, 41, 47, 48, 49, 53, 55, 59, 61, 62, 63

[162] G.-H. Liu, L. Zhang, Y.-K. Hou, Z.-Y. Li, and J.-Y. Yang. Image retrieval based on multi-texton histogram. *Pattern Recognition*, 43(7):2380–2389, 2010. → page 193

[163] Y. Liu and Q. Dai. A survey of computer vision applied in aerial robotic vehicles. In *2010 International Conference on Optics, Photonics and Energy Engineering (OPEE)*, volume 1, pages 277–280, 2010. → page 12

[164] N. V. Lopes, P. A. M. do Couto, H. Bustince, and P. Melo-Pinto. Automatic histogram threshold using fuzzy measures. *IEEE Transactions on Image Processing*, 19(1):199–204, 2010. → page 193

[165] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, pages 227–236, 2008. → page 182

[166] A. Luppold, D. Oehlert, and H. Falk. Evaluating the performance of solvers for integer-linear programming. Technical report, 2018. → page 173

[167] K. Mack, E. Cutrell, B. Lee, and M. R. Morris. Designing tools for high-quality alt text authoring. In *The 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, pages 1–14, 2021. → page 175

[168] S. Mahajan and W. G. Halfond. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proc. of ASE '14*, pages 91–96, 2014. → pages 10, 29, 34, 40, 41, 46, 48, 49, 52, 55, 60, 61

[169] S. Mahajan and W. G. J. Halfond. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *Proc. of ICST '15*, pages 1–10, 2015. → pages 29, 34, 40, 41, 46, 48, 49, 52, 55, 59, 60, 61, 66

[170] 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. → page 104

[171] S. Mahajan, B. Li, P. Behnamghader, and W. G. J. Halfond. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *Proc. of ICST '16*, ICST '16, pages 191–201, 2016. → pages 29, 34, 40, 41, 46, 48, 49, 52, 55, 60, 61

[172] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014. URL http://www.aclweb.org/anthology/P/P14/P14-5010. → page 128

[173] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation

algorithms and measuring ecological statistics. In *Proc. of ICCV '01*, pages 416–423, 2001. → page 67

[174] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. Le Traon. Automating the extraction of model-based software product lines from model variants (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 396–406. IEEE, 2015. → page 212

[175] J. Matas, C. Galambos, and J. Kittler. Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding*, 78(1):119–137, 2000. → page 83

[176] D. Mazinanian and N. Tsantalis. Migrating Cascading Style Sheets to Preprocessors by Introducing Mixins. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2016, pages 672–683, 2016. → page 213

[177] D. Mazinanian and N. Tsantalis. CSSDev: Refactoring duplication in Cascading Style Sheets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE) Companion*, ICSE 2017, 2017. → page 213

[178] D. Mazinanian, N. Tsantalis, and A. Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 496–506, 2014. ISBN 978-1-4503-3056-5. → page 213

[179] P. B. Meggs. *Type and Image: The Language of Graphic Design*. Van Nostrand Reinhold, 1992. ISBN 0442011652. → page 183

[180] A. Mesbah and A. van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE Computer Society, 2007. → page 212

[181] B. Mihalcea. User interface construction with mockup images, Feb. 11 2014. US Patent 8,650,503. → page 180

[182] 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*, 29(3):381, 2013. doi: 10.1093/bioinformatics/bts677. URL +http://dx.doi.org/10.1093/bioinformatics/bts677. → pages 95, 101

[183] Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. Vizmod tool repository, 2018. URL https://github.com/msbajammal/vizmod. → pages 128, 140, 146, 164, 165, 174, 200, 211

[184] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. *Applied Computing Review*, 12(3):20–36, Sept. 2012. ISSN 1559-6915. → page 182

[185] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk. Automated reporting of gui design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 165–175, 2018. → pages 29, 40, 48, 52, 53, 61

[186] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk. Detecting and summarizing gui changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 543–553, 2018. → pages 29, 40, 48, 52, 61

[187] K. P. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, 2018. → pages 29, 40, 48, 53, 54, 55, 61

[188] L. Moreno, M. González, and P. Martínez. Captcha and accessibility. *Is this the best we can do*, 2014. → page 169

[189] Mozilla Developer Network. Web components, 2017. URL https://developer.mozilla.org/en-US/docs/Web/Web_Components. Accessed: 4 April 2018. → pages 195, 209

[190] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202. ACM, 1992. → pages 1, 179

[191] S. Natarajan and C. Csallner. P2a: A tool for converting pixels to animated mobile application user interfaces. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 224–235. ACM, 2018. → pages 29, 40, 48, 61

[192] M. W. Newman and J. A. Landay. Sitemaps, storyboards, and specifications: A sketch of web site design practice. In *Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices,*

Methods, and Techniques (DIS), pages 263–274, 2000. ISBN 1-58113-219-0. → page 179

[193] T. A. Nguyen and C. Csallner. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *Proc. of ASE '15*, pages 248–259, 2015. → pages 29, 37, 38, 40, 46, 48, 50, 53, 55, 59, 61, 62

[194] M. Nixon and A. Aguado. *Feature extraction and image processing for computer vision*. Academic Press, 2019. → page 118

[195] M. Noorjahan. A bio metric based approach for using captcha–to enhance accessibility for the visually impaired. *Disability and Rehabilitation: Assistive Technology*, 2019. → page 169

[196] R. L. Novais, A. Torres, T. S. Mendes, M. Mendonça, and N. Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 2013. → page 15

[197] C. L. Novak and S. A. Shafer. Anatomy of a color histogram. In *Proceedings 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 599–605, June 1992. doi: 10.1109/CVPR.1992.223129. → page 52

[198] M. H. Osman, T. Ho-Quang, and M. Chaudron. An automated approach for classifying reverse-engineered and forward-engineered uml class diagrams. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 396–399. IEEE, 2018. → pages 29, 40, 48, 61

[199] F. K. Ozenc, M. Kim, J. Zimmerman, S. Oney, and B. Myers. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 2513–2522, 2010. ISBN 978-1-60558-929-9. → page 179

[200] K. Park, T. Goh, and H.-J. So. Toward accessible mobile application design: developing mobile application accessibility guidelines for people with visual impairment. *Proceedings of HCI Korea*, pages 31–38, 2014. → page 141

[201] N. Patil, D. Bhole, and P. Shete. Enhanced ui automator viewer with improved android accessibility evaluation features. In *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*, pages 977–983. IEEE, 2016. → pages 141, 176

[202] M. Patrick, M. D. Castle, R. O. J. H. Stutt, and C. A. Gilligan. Automatic Test Image Generation Using Procedural Noise. In *Proc. of ASE '16*, pages 654–659, 2016. → pages 29, 35, 40, 46, 48, 50, 53, 59, 60, 61

[203] L. Ponzanelli, G. Bavota, A. Mocci, M. D. Penta, R. Oliveto, M. Hasan, B. Russo, S. Haiduc, and M. Lanza. Too Long; Didn't Watch! Extracting Relevant Fragments from Software Development Video Tutorials. In *Proc. of ICSE '16*, pages 261–272, 2016. → pages 29, 38, 40, 47, 48, 50, 52, 53, 55, 60, 61, 62, 63

[204] Publications Office of the European Union. Directive (eu) 2016/2102 of the european parliament on the accessibility of the websites and mobile applications of public sector bodies. https://eur-lex.europa.eu/eli/dir/2016/2102/oj, 2016. → pages 6, 106

[205] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In *Proceedings of the 5th International Conference of Web Engineering (ICWE)*, pages 252–262, 2005. → page 212

[206] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 116–126, 2007. ISBN 0-7695-2828-7. → page 212

[207] Ó. S. Ramón, J. S. Cuadrado, J. G. Molina, and J. Vanderdonckt. A layout inference algorithm for graphical user interfaces. *Information and Software Technology*, 70:155–175, 2016. → page 180

[208] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199, 2013. ISSN 0950-5849. → page 212

[209] React Issues on GitHub. White-space between inline elements #1643, 2017. URL https://github.com/facebook/react/issues/1643. Accessed: 16 April 2018. → page 202

[210] ReactJS. Thinking in components, 2018. URL https://reactjs.org/docs/thinking-in-react.html. Accessed: 15 February 2018. → pages 8, 180, 183

[211] K. Reinecke, D. R. Flatla, and C. Brooks. Enabling designers to foresee which colors users cannot see. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 2693–2704, 2016. → pages 29, 36, 40, 48, 61

[212] S. P. Reiss and Q. Miao, Yun Xin. Seeking the user interface. *ASE*, 25(1): 157–193, 2018. → pages 29, 38, 40, 47, 48, 50, 59, 61, 62

[213] repository for ICST 2018: . https://github.com/msbajammal/canvasure. → page v

[214] C. Roberts, G. Wakefield, and M. Wright. The web browser as synthesizer and interface. In *NIME*, pages 313–318. Citeseer, 2013. → pages 95, 101

[215] I. Rodriguez, L. Llana, and P. Rabanal. A general testability theory: Classes, properties, complexity, and testing reductions. *IEEE Transactions on software engineering*, 40(9):862–894, 2014. → page 5

[216] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock. Examining image-based button labeling for accessibility in android apps through large-scale analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 119–130, 2018. → pages 141, 176

[217] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen's School of Computing, 2007. → page 212

[218] C. K. Roy and J. R. Cordy. NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008. ISBN 978-0-7695-3176-2. → pages 184, 212

[219] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009. ISSN 0167-6423. → page 212

[220] S. Roy Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proc. of ICSE '13*, pages 702–711, 2013. → pages 10, 29, 34, 40, 41, 42, 46, 48, 49, 52, 55, 59, 60, 61, 62

[221] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, 2015. → page 67

[222] J.-B. Ryu, C.-G. Lee, and H.-H. Park. Formula for harris corner detector. *Electronics letters*, 47(3):180–181, 2011. → page 88

[223] L. N. Sabaren, M. A. Mascheroni, C. L. Greiner, and E. Irrazábal. A systematic literature review in cross-browser testing. *JCST*, 18(01), Apr. 2018. → pages 13, 14, 34

[224] A. Sadeghi, R. Jabbarvand, and S. Malek. Patdroid: permission-aware gui testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 220–232, 2017. → page 177

[225] S. Sanchez-Gordon and S. Luján-Mora. A method for accessibility testing of web applications in agile environments. In *Proceedings of the 7th World Congress for Software Quality (WCSQ). En proceso de publicación.(citado en la página 13, 15, 85)*, 2017. → pages 141, 176

[226] SAP S.E. Accessibility testing software automation tool, 2019. US Patent 10,275,339. → pages 107, 145

[227] A. Scharf and T. Amma. Dynamic Injection of Sketching Features into GEF Based Diagram Editors. In *Proc. of ICSE '13*, pages 822–831, 2013. → pages 10, 29, 36, 37, 40, 47, 48, 50, 60, 61, 64

[228] H. Scharr. *Optimal operators in digital image processing*. PhD thesis, 2000. → page 80

[229] J. Seifert, B. Pfleging, E. del Carmen Valderrama Bahamóndez, M. Hermes, E. Rukzio, and A. Schmidt. Mobidev: A tool for creating apps on mobile phones. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 109–112. ACM, 2011. → pages 29, 40, 48

[230] E. Selay, Z. Q. Zhou, and J. Zou. Adaptive Random Testing for Image Comparison in Regression Web Testing. In *Proc. of DICTA '14*, DICTA '14, pages 1–7, 2014. → pages 10, 29, 34, 40, 41, 46, 48, 49, 59, 61

[231] N. Semenenko, M. Dumas, and T. Saar. Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features. In *Proc. of ICSM '13*, pages 528–531, 2013. → pages 10, 29, 34, 40, 41, 42, 46, 48, 50, 53, 54, 59, 61

[232] L. C. Serra, L. P. Carvalho, L. P. Ferreira, J. B. S. Vaz, and A. P. Freire. Accessibility evaluation of e-government mobile applications in brazil. *Procedia Computer Science*, 67:348–357, 2015. → page 141

[233] Seyfarth Shaw Law Firm, LLP. Americans with disabilities (ada) act, title iii report. https://www.seyfarth.com/news-publications, 2018. → pages xv, 109, 145

[234] M. Sezgin et al. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic imaging*, 13(1): 146–168, 2004. → page 81

[235] C. Shen and Q. Zhao. Webpage saliency. In *European conference on computer vision*, pages 33–46. Springer, 2014. → page 152

[236] A. U. Sinha and S. A. Armstrong. iCanPlot: Visual exploration of high-throughput omics data using interactive canvas plotting. *PLoS ONE*, 7 (2), 2012. → pages 95, 101

[237] N. Sinha and R. Karim. Compiling Mockups to Flexible UIs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 312–322, 2013. ISBN 978-1-4503-2237-9. → page 180

[238] S. Snider, W. L. Scott II, and S. Trewin. Accessibility information needs in the enterprise. *ACM Transactions on Accessible Computing (TACCESS)*, 12(4):1–23, 2020. → pages 141, 145, 176

[239] I. C. Society, P. Bourque, and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. 3rd edition, 2014. → page 16

[240] W. Song, X. Qian, and J. Huang. Ehbdroid: beyond gui testing for android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 27–37. IEEE, 2017. → page 177

[241] J. H. Sossa-Azuela et al. On the computation of the euler number of a binary object. *Pattern recognition*, 29(3):471–476, 1996. → page 82

[242] StackOverflow. Developer survey results, 2017. URL https://insights.stackoverflow.com/survey/2017. Accessed: 4 April 2018. → pages 195, 209

[243] stateofjs.com. Worldwide usage of javascript front-end libraries, 2018. URL https://stateofjs.com/2017/front-end/results/. Accessed: 10 April 2018. → pages 8, 180, 182, 195, 209

[244] A. Stocco, R. Yandrapally, and A. Mesbah. Visual web test repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018. ACM, 2018. → page 212

[245] A. Stocco, R. Yandrapally, and A. Mesbah. Visual web test repair. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 12 pages, 2018. → pages 10, 29, 37, 40, 43, 48, 53, 54, 55, 61

[246] M.-A. D. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202. ACM, 2005. → page 15

[247] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017. → page 176

[248] S.-H. Sun, H. Noh, S. Somasundaram, and J. Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pages 4790–4799, 2018. → pages 29, 38, 40, 48, 61

[249] A. Swearngin and Y. Li. Modeling mobile interface tappability using crowdsourcing and deep learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019. → pages 29, 36, 40, 48, 61

[250] N. Synytskyy, J. R. Cordy, and T. R. Dean. Resolution of static clones in dynamic Web pages. In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution (WSE)*, pages 49–56, 2003. ISBN 0-7695-2016-2. → page 212

[251] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision, 2015. → page 125

[252] H. Tanno and Y. Adachi. Support for finding presentation failures by using computer vision techniques. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 356–363, 2018. → pages 29, 40, 43, 48, 53, 55, 61

[253] Title 42 of the United States Code, Section 12101. The americans with disabilities act. https://www.law.cornell.edu/uscode/text/42/12101, 2020. → pages 6, 106

[254] M. S. Tooms. *Colour Reproduction in Electronic Imaging Systems: Photography, Television, Cinematography*. John Wiley & Sons, 2016. → page 79

[255] TSE 2020 repository: . http://tiny.cc/tse-2020. → page v

[256] United Kingdom Government's Office of Digital Services. Accessibility tools audit. https://alphagov.github.io/accessibility-tool-audit/, 2018. → pages 3, 6, 107, 113, 139, 166, 175

[257] United States Courts, Court Records. Public access to court electronic records (pacer). https://www.uscourts.gov/court-records, 2020. → pages xv, 109, 145

[258] D. van der Linden and I. Hadar. A systematic literature review of applications of the physics of notations. *IEEE Transactions on Software Engineering*, 45(8):736–759, 2019. → page 13

[259] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez. Can everyone use my app? an empirical study on accessibility in android apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–52. IEEE, 2019. → pages 106, 145

[260] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim, W. Aigner, R. Borgo, F. Ganovelli, and I. Viola. A survey of visualization systems for malware analysis. In *EuroVis (STARs)*, pages 105–125, 2015. → page 14

[261] S. v. d. Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. → pages 93, 200

[262] M. Wan, Y. Jin, D. Jin, J. Gui, S. Mahajan, and W. G. J. Halfond. Detecting display energy hotspots in android apps. volume 27, 2017. → pages 29, 34, 38, 40, 46, 48, 50, 53, 59, 61, 63

[263] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*

(ESEC/FSE), pages 455–465, 2013. ISBN 978-1-4503-2237-9. → page 184

[264] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. → page 52

[265] WebAIM Non-profit Organization. Web accessibility evaluation tool. URL https://wave.webaim.org/. → page 141

[266] WebAIM Non-profit Organization. Screen reader user survey results., 2019. URL https://webaim.org/projects/screenreadersurvey8/. → pages 6, 107, 109, 112, 114, 126, 141

[267] O. Westesson, M. Skinner, and I. Holmes. Visualizing next-generation sequencing data with jbrowse. *Briefings in Bioinformatics*, 14(2):172, 2013. doi: 10.1093/bib/bbr078. URL +http://dx.doi.org/10.1093/bib/bbr078. → pages 95, 101

[268] W. E. Winkler. Overview of record linkage and current research directions. In *Bureau of the Census*. Citeseer, 2006. → page 98

[269] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proc. of EASE '14*, pages 1–10, 2014. → page 26

[270] L. Wroblewski. *Web form design: filling in the blanks*. Rosenfeld Media, 2008. → page 153

[271] S. Wu, J. Wieland, O. Farivar, and J. Schiller. Automatic alt-text: Computer-generated image descriptions for blind users on a social network service. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 1180–1192, 2017. → pages 29, 38, 40, 48, 61

[272] S. Wu, J. Wieland, O. Farivar, and J. Schiller. Automatic alt-text: Computer-generated image descriptions for blind users on a social network service. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 1180–1192, 2017. → page 175

[273] Z. Wu, Y. Jiang, Y. Liu, and X. Ma. Predicting and diagnosing user engagement with mobile ui animation via a data-driven approach. In

Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pages 1–13, 2020. → pages 29, 36, 40, 48, 61

[274] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao. Automatic identification of sensitive ui widgets based on icon classification for android apps. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, ICSE 2019, 2019. → pages 29, 40, 48, 53, 54, 55, 61

[275] Z. Xu and J. Miller. Cross-Browser Differences Detection Based on an Empirical Metric for Web Page Visual Similarity. *TOIT*, 18(3):1–23, 2018. → pages 29, 34, 40, 41, 46, 48, 49, 52, 59, 60, 61

[276] S. Yan and P. Ramachandran. The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing (TACCESS)*, 12(1): 1–31, 2019. → pages 141, 176

[277] R. Yandrapally, A. Stocco, and A. Mesbah. Near-duplicate detection in web app model inference. In *Proceedings of 42nd International Conference on Software Engineering*, ICSE '20, page 12 pages. ACM, 2020. → page 66

[278] B. Yang, F. Gu, and X. Niu. Block mean value based image perceptual hashing. In *2006 International Conference on Intelligent Information Hiding and Multimedia*, pages 167–172. IEEE, 2006. → pages 48, 52

[279] H. Yee, S. Pattanaik, and D. P. Greenberg. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Transactions on Graphics (TOG)*, 20(1):39–65, 2001. → pages 48, 52

[280] Y. Yesilada and S. Harper. *Web Accessibility: A Foundation for Research*. Springer, 2019. → pages 145, 166, 174

[281] S. Yu, C. Fang, Y. Feng, W. Zhao, and Z. Chen. Lirat: layout and image recognition driving automated mobile testing of cross-platform. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1066–1069. IEEE, 2019. → pages 29, 40, 48, 55, 61

[282] A. Yuan and Y. Li. Modeling human visual search performance on realistic webpages using analytical and deep learning methods. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020. → pages 29, 35, 40, 48, 61

[283] C. Zhang, H. Cheng, E. Tang, X. Chen, L. Bu, and X. Li. Sketch-guided GUI Test Generation for Mobile Applications. In *Proc. of ASE '17*, pages 38–43, 2017. → pages 29, 35, 40, 41, 44, 46, 48, 50, 61

[284] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. → page 13

[285] Y. Zhang, Y. Xiao, M. Chen, J. Zhang, and H. Deng. A survey of security visualization for computer network logs. *Security and Communication Networks*, 5(4):404–421, 2012. → page 14

[286] D. Zhao, Z. Xing, C. Chen, X. Xia, and G. Li. Actionnet: vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 350–361. IEEE, 2019. → pages 29, 40, 48, 55, 61

[287] X. S. Zheng, I. Chakraborty, J. J.-W. Lin, and R. Rauschenberger. Correlating low-level image statistics with users-rapid aesthetic and affective judgments of web pages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1–10, 2009. → pages 29, 35, 40, 48, 61