Challenges for Layout Validation: Lessons Learned

Santiago Bragagnolo, Benoît Verhaeghe, Abderrahmane Seriai, Mustapha Derras, Anne Etien

Challenges for Layout Validation: Lessons Learned

Santiago Bragagnolo[1,2], Benoît Verhaeghe[1,2], Abderrahmane Seriai[1], Mustapha Derras[1], and Anne Etien[2]

[1] Berger-Levrault, Montpellier, France
{firstname}.{lastname}@berger-levrault.com
[2] Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France
{firstname}.{lastname}@inria.fr

Abstract. Companies are migrating their software systems. The migration process contemplates many steps, UI migration is one of them. To validate the UI migration, most existing approaches rely on visual structure (DOM) comparison. However, in previous work, we experimented such validation and reported that it is not sufficient to ensure a result that is equivalent or even identical to the visual structure of the interface to be migrated. Indeed, two similar DOM may be rendered completely differently. So, we decide to focus on the layout migration validation. We propose a first visual comparison approach for migrated layout validation and experiment it on an industrial case. Hence, from this first experiment and already existing studies on image comparison field, we highlight challenges for layout comparison. For each challenge, we propose possible solutions, and we detail the three main features we need to create a good layout validation approach.

Keywords: GUI Migration · Challenges · Comparison · Validation

1 Introduction

With the fast evolution of programming languages and frameworks, companies tend to update their software more and more. This evolution may imply the migration of their application GUI [15]. To ensure the proper software operation after the update, one needs to validate and ensure the correct the migration of GUI. Whereas manual validation is always possible, it is tedious, error-prone, time-consuming, and is expensive for the companies. So we look for automatic validation approaches.

While approaches base their validation on DOM[3] comparison [4], few discute the validation of the rendered UI. The visual aspect of an application is mostly neglected, although it is essential for the end-user of the application [11], and thus to the acceptance of the new software. Since the software may also be accepted or rejected by its look and feel, we consider that UI validation is extremely important for the success of the migration.

[3] Document Object Model

A migration process has one of two different objectives in relation to the migration of the UI: such process is rather **visually constant**, or **layout constant**. In the case where a migration process is visually constant, it aims to produce a migrated version with identical UI, from the layout of the widgets , to the look and feel of the widgets. In the other hand, when the migration process is layout constant, it aims to produce an enriched migrated version with the same layout, but possibly visually different widgets. In both cases, the validation of the migrated layout is a first step to the UI validation.

Inspired by other research fields [3, 13, 14], we propose an approach to compare the layout of migrated applications with the original layouts. We experimented with this approach on a real industrial migration project. From this experience, we report a list of challenges for layout validation and provide some solutions.

In the further sections, we discuss the need for validation in general and particularly about layout validation (Section 2). We present the different existing approaches to tackle down this problematics (Section 3), to explain the position of our solution. We draft our validation process (Section 4), and give place to the core of this article, the report of challenges (Section 5), where we describe each of the problematics we found on the development of our method. We identify the features that can help to solve those challenges (Section 6), and after a conclusion, we present the middle term goals of our work (Section 7).

2 UI Validation

Our work takes place in collaboration with Berger-Levrault[4], a major IT company selling Web applications developed in GWT. Unfortunately, GWT is no longer maintained and the last update was made in 2015. As a consequence, Berger-Levrault decided to migrate its applications to Angular 6. This migration is crucial since Berger-Levrault has more than 8 applications in GWT each including more than 500 web pages.

In preceding work [15], we proposed an approach to migrate the front-end of applications. We implemented this approach to migrate the GWT applications of Berger-Levrault to Angular. Once the migration is performed, we need to validate that the applications are correctly migrated.

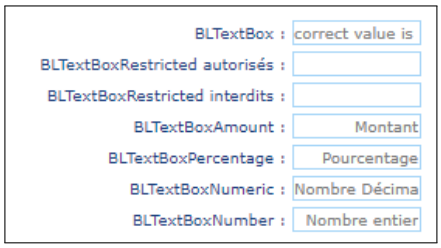
Migration and validation are part of the same process. Once the validation is done, the results are going to be used for enhancing the migration and fixing errors. This new migration has to be validated once again, triggering a new process of migration. This loop recurs again until the process of migration is finished. In this context, manual validation for each iteration of the migration process is expensive in terms of money and time. Hence, we propose to rely on automatic validation.

In this section we present the main migration validation approach we tried from the literature and detail why it is not sufficient (Section 2.1). Then we detail what is a layout validation (Section 2.2).

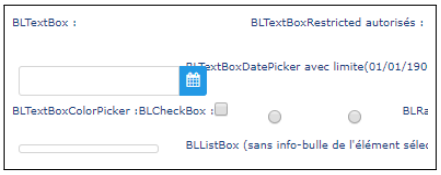
[4] https://www.berger-levrault.com

2.1 Current Migration Validation Approach

In the experimentations of validation of UI we started trying to use the common means proposed by the literature. Joorabchi and Mesbah [5], Memon et al. [7] and Sánchez Ramón et al. [11] defined metrics to verify the success of the migration process. They checked that all the widgets and attributes are detected by their tools. Each widget and attribute must be identified and reachable, which means the entity type must be discovered, migrated, and present in the target applications. Each widget should also belong to the right container and its attributes created with the right value.



(a) GWT original



(b) Angular migration

Fig. 1: Migration of a page without considering the layout

We experimented with the above validation on the case of Berger-Levrault. Despite that, it reported good results according to the proposed metrics, the origin and generated UIs were completely different. Figure 1 presents the migration of one web page of a Berger-Levrault application. On the left-hand side (Figure 1a), it shows the original page, and on the right-hand side (Figure 1b) the page after the migration.

The traditional DOM proposes a tree as containing structure, where we can have elements composed by other elements, defining a strong relation of containment and scoping. Like this we can define a document with header, body and footer. Each of these parts can hold internal divisions, sections as well as widgets and components, recursively contained. Comparing DOM expecting to have a direct implication on the page rendering is the first solution. Two pages (the original and the migrated ones) may have equivalent DOM and thus plainly satisfy the proposed metrics. However, the migrated UI (Figure 1b) and the original version (Figure 1a) do not have much in common to the human eye.

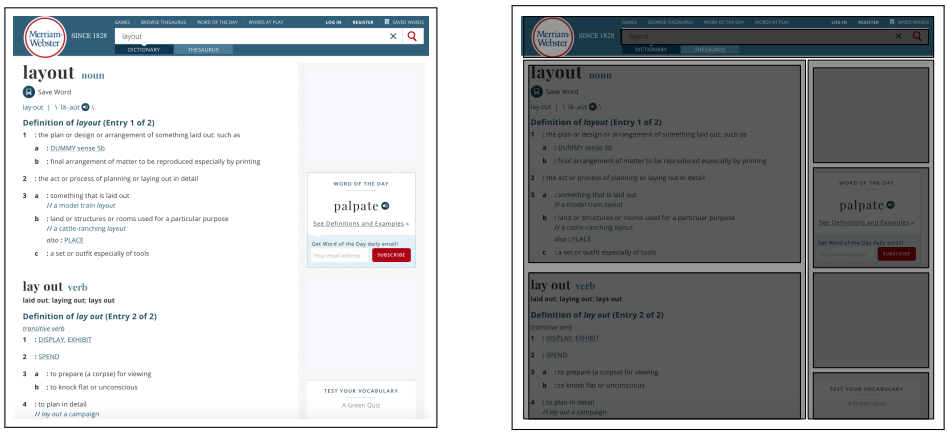
Comparing DOM might be a good starting point to compare pages but it is, now-a-days, certainly not sufficient. In modern applications, the layout and style are managed orthogonally to the DOM composition. Thus, these approaches are not suitable for validating modern applications.

This is why we propose to add another dimension to the validation of the UI migration: the layout.

2.2 Layout Validation

From Merriam&Webster dictionary[5] a Layout is *the plan or design or arrangement of something laid out.*

From this definition, we consider that the layout is the position of the UI elements the one against the others. In a UI, we have two main kinds of components. Those that contain other components, for defining groups of components, and those that are contained. The components containing others, such as field set, card, panels, etc., are responsible for defining the main disposition of the contained elements in the page. For this reason we call those UI elements: "structural layout elements".



(a) Page (b) Page's structural layout elements

Fig. 2: A layout example

In Figure 2, we present the relationship between the UI and its layout. Figure 2a shows the page as interpreted by the browser. And, in Figure 2b we highlighted the structural layout elements of the showed content. By highlighting these structural layout elements we are thus revealing the layout. The empty layout boxes belong to adds that have been silenced by the usage of ad-blocking plugins.

As we pointed out previously in Section 1, the migration may be required to be visually constant or layout constant. To validate that the migrated UI complies with the UI expectations we must take into account at least the layout.

3 State of the Art

To compare the rendered UI after performing a migration, several solutions can be considered regarding the approaches existing in the literature. First, in Section

[5] https://www.merriam-webster.com/dictionary/layout

3.1, we expose existing validation approaches used to compare the visual aspect of two software systems. Second, in Section 3.2, we present approaches from other research fields that are related to image comparisons and which we think we can use in our domain.

3.1 Applications Comparison

Moran et al. [9] compared the UI of android applications. They proposed an approach to detect GUI changes in evolving mobile applications (*e.g.,* between two versions of the same application). Their approach has two main concerns: mapping the screens between the applications version (*i.e.,* which previous screen corresponds to which actual screen), and detecting the GUI changes. For the change detection, they rely on a pixel by pixel comparison of the screenshots of the previous and current applications.

Sánchez Ramón et al. [12] proposed an approach to infer a hierarchical layout from a UI with hardcoded widgets positions. To retrieve this layout, they use the closeness metric between two widgets. This metric allows them to compute the visual proximity of two elements in a UI. By grouping widgets together, they create the new layout definition.

Cao et al. [3] migrated web archives from HTML4 to HTML5. To validate the migration they proposed to segment the original and the migrated pages in blocks using the DOM. Then, they took screenshots of the original and migrated applications with blocks and computed the differences between the two pages.

Sanoja and Gançarski [13] proposed a segmentation method for web page analysis. Their method consists on dividing a web page into blocks. To retrieve the blocks, CSS and HTML provide them the position of all widgets on the web page, and they use background space to separate two blocks. For example, if there is no space between two elements, they are considered in the same block.

Alpuente and Romero [1] proposed UI comparison based on DOM analysis. Since the observation that two different DOM may render the same UI, they proposed to infer the visual structure (*i.e.,* tree) of a web application from its DOM. To do so, they classified the HTML tags into four categories, the group, the row, the col, and the text. Then, they translated the HTML DOM using their terminology, and they compressed the new UI tree. The compression corresponds to a simplification of the new tree, for example, by grouping two groups together. They considered two different pages having the same visual tree as visually similar.

The authors proposed different ways to represents layouts, infer them, and compare them. Some approaches rely on DOM information and DOM comparison while others rely on screenshot comparison. Except [9] and [1], comparison approaches use blocks as a way to simplify the comparison problem and focus on the layout aspect. The blocks were created from DOM information.

3.2 Images Comparison

Another strategy to compare images is to take inspiration from the work on image retrieval. This field is focused on determining if an image is similar to or contains another one.

Van Beusekom et al. [14] proposed an approach to retrieve images based on their layouts. To do so, they extracted from each image its layout. The structural layout elements are represented by blocks. Then, to compute the distance between two images, the authors compute the distance between the layouts, and so between the blocks. To improve their result, they also match each block of an image with the other block of the compared image.

Finally, the image comparison approaches [6, 10] are used to identify image similarity. This approach allows one to compare two images and find if an image is present in the other. It can be used to determine if two images are originally identical even after distortion or rotation, or to determine if part of an image is present in another one.

The proposed approaches might be used to compare two screenshots. However, since those approaches have been designed to retrieve an image inside another it is different from our problem. So, further work must be done to verify if their results are relevant in our context.

4 First Sketch of Solution

To validate the migration of the layout, *i.e.,* the identical positioning of UI elements the ones against the others, we proposed and implemented an approach[6].

Our approach aims to highlight the structural layout elements. It is inspired by the definition of blocks proposed by Sanoja and Gançarski [13], Cao et al. [3] and Van Beusekom et al. [14] The approach is divided into five steps.

First step: detecting pages to validate. The first step concerns the original and the target applications. For each of them, this step consists in detecting all the pages for which we have to validate the migration. By detection, we mean being able to reach given pages if the list is known or crawling the full application in the opposite case. Reaching a page is rather trivial, in traditional web development approaches; by precising the related URL. However, it becomes complex, for example in modern single page applications (from now on, SPA)[7], where different components are accessed not by using URL, but by applying specific flows of user interactions, *e.g.,* click, double click, hover, etc. The output of this step is a list of pages of the source application to validate, the way to access them, their corresponding pages in the migrated application as well as the way to access these latter. There are different techniques of crawling and discovery that suit this case. In the context of Berger-Levrault, we rely on the migration

[6] https://github.com/badetitou/Pasino

[7] A Single page application is a web application or website that interacts with the web browser by dynamically rewriting the current web page

tool that provides us this information gathered during the process of migration.

The four next steps are iteratively repeated for each couple of pages (one of the source application and its corresponding one in the migrated application). We describe the next step for a couple of source and migrated pages.

Second step: browsing original and migrated pages. The two pages are browsed by using a browser (*i.e.,*Firefox, Chrome, Edge, Safari, etc.). The same issues are faced concerning SPA applications what is the case for Berger-Levrault applications. In that cases, we use Selenium[8] to navigate through pages by simulating user interactions and access to the page to analyze.

Third step: creating blocks. Each browsed page must be prepared and normalized for further comparisons, *i.e.,* the size of the pages must be the same before taking screenshots to ease future image comparison. In this step, we choose and extract the elements to compare. Since we are validating layouts, we must emphasize the structural layout elements with their inner structural layout elements. Since we are not validating the look of the components, such as buttons, labels, text boxes, etc., we must underemphasize or silence the content for ignoring the comparison of these details. Concretely, we *create blocks* corresponding to each of the structural layout elements. In the context of Berger-Levrault, we apply a new CSS on the pages. The CSS converts all fieldset widgets into blocks with transparency. Thus, we can look at blocks composition.

Fourth step: taking screenshot. Our approach relies on a visual validation. Consequently, after creating the blocks, we take snapshot of the result as a visual mean of comparison. So we get an image with only structural layout elements.

Fifth step: comparing. We compare, pixel by pixel, the screenshots of the source application and the ones of the migrated application.

(a) GWT original page

(b) Angular migrated page

(c) Blocks GWT page

(d) Blocks Angular page

Fig. 3: Apply approach on GWT to Angular migration

[8] https://www.selenium.dev/

We applied our tool on the migration project of Berger-Levrault from GWT to Angular. Figure 3 shows the screenshots created by our approach using blocks. On the left-hand side, it shows the original page screenshot and its corresponding screenshot after applying the creation of blocks. On the right-hand side, it shows the equivalent screenshots for the migrated page.

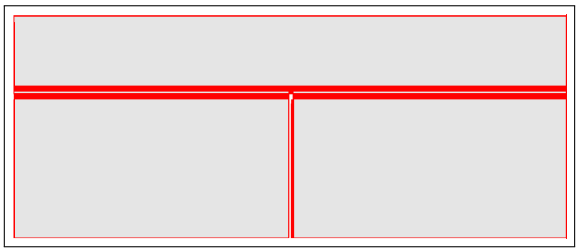
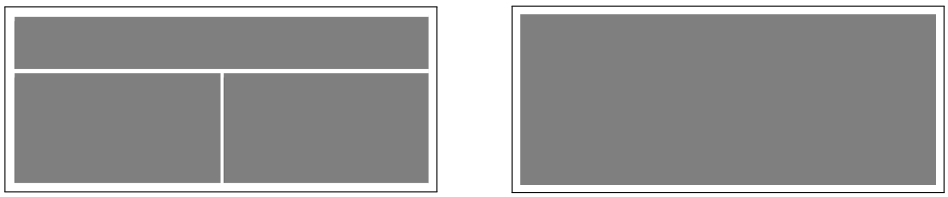


Fig. 4: *Diff* between pages

Although it looks visually equivalent, and there are no differences between the two, the distance between the blocks and the size of the blocks are slightly different. Figure 4 shows the difference pixel by pixel of the blocks screenshots. Red pixels represent positions where there are differences between the source and the migrated pages. Even though there are few perceptual differences between the two images, the comparison of the blocks reports 9% of the exported image incorrectly migrated.



(a) Blocks GWT page (b) Angular Grey box

Fig. 5: Difference in between original and a full grey block

This 9% may look like a small number. In our context, this is not true. Indeed, Figure 5 shows pixel by pixel comparison, between the original page and a full grey block of the same size. As can be seen in Figure 6, following the same strategy to measure the difference between the two screenshots, it was reported 5% of the image bad exported. So, our strategy reports that a full grey block layout is better than the one created from a real migration. But, it is completely false. Thus, it confirms that we need a smarter way to validate layout migration.

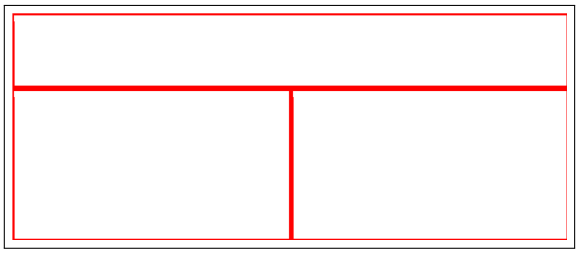


Fig. 6: *Diff* between original and a full grey block

5 Challenges of Layout Validation

From our experiment and the state of the art, we identified several challenges for the layout validation. Those challenges have to be considered for future layout validation tools. We identified 6 challenges: structural layout elements (Section 5.1), Ajax-based architecture (Section 5.2), successive shifting (Section 5.3), dynamic content (Section 5.4), interactive widget (Section 5.5), and overlap (Section 5.6).

5.1 Structural Layout Elements

Problem: One major problem is how to identify the structural layout elements in a page. In our experiment, we considered that fieldsets are the structural layout elements of all pages. However, layout also exists in pages where the DOM does not contain any fieldsets. So, one challenge is to define how the layout is expressed.

Solution: We identified two ways to solve this problem. One is to rely on DOM and CSS information. For each source language, we need to manually define what are the structural elements. For example, it can be CSS classes in modern web applications; or table nesting in legacy systems, as GWT. The other solution is to extract the structural layout by analyzing the screenshots [13].

5.2 Ajax-based Architecture

Problem: One requirement to validate the UI migration is to be able to browse each couple of source and migrated UI pages. One can think, for a web application, of using URL of each page as a reference. However, recent web applications are developed using the Ajax framework. Ajax allows developers to modify the UI of a page without properly navigating, *e.g.,* changing the URL. It is the case of SPA promoted by recent UI frameworks: Angular, React, etc. So to browse a page, a tool can not simply get the URL content but need to perform actions on the UI.

Solution: To browse each page, the validation tool needs to know the suite of actions to perform, and a way to execute them. The suite of actions can be extracted using a crawler [2], however, crawling SPA application is complex [8]. Then, to perform the actions, we propose to use already developed tools used in GUI testing such as Selenium. Those tools allow one to programs interaction with a UI.

5.3 Successive Shifting

Problem: As identified by Sanoja and Gançarski [13], the shifting of one block (because it is rendered with an incorrect size or position) may cause shifting of other blocks. Moreover, a slight error repeated on each block (for example each block is larger by only a few pixels) may create important differences in a screenshot but impact only slightly the layout.

Solution: Instead of comparing an image pixel by pixel or block position by block position, one can compare the position of blocks relative to visually near blocks. Thus, the validation approach will report minimal differences and not completely different pages. This comparison is more complex because it requires block identification, *i.e.,* recognizing the same blocks in source and migrated applications.

5.4 Dynamic Content

Problem: Some widgets, such as a table, can display information coming from an external server. If the received data changed, or if the component does not receive the data, the widget does not fill the same space in the original and migrated UI. While missing data does not impact the layout definition (in terms of relationships), it impacts the pixel to pixel comparison.

Solution: We found two ways to solve this problem. One is to identify the blocks in the original and migrated UI, then to compute the relationship between blocks. If the relationships are the same in the original and migrated UI, then the UI have the same layout. The other is to empty out the dynamic components applying some javascript routines, and thus do not consider data but still the default size of the component.

5.5 Interactive Widget

Problem: Some widgets are interactive. It is the case of the expandable panel, a panel that can be *opened* or *closed* by the user. The state of such components does not impact the layout but can change the size of the blocks. Thus, in block to block or pixel to pixel comparison, the validation tool reports bad migration whereas for example, states are not the same, by default in both applications.

Solution: One solution is to collect the state of the widgets in the original application, and then to set the state of the widgets in the migrated application before taking the screenshot. To set the states, one can use a tool such as Selenium.

5.6 Overlap

Problem: User interfaces are composed of multiple structural layout elements, *i.e.,* panel, fieldset, card. Proposed approaches, like ours, validate the layout migration by comparing layout composing blocks. Such approaches must consider that some layout elements overlap other layout elements. So, the z-index, *i.e.,* defining which widget is rendered on top of which one, must be extracted to validate correctly the interface.

Solution: One solution to handle the overlap is to use transparency when displaying the blocks. This solves the problem of a block inside another, but it does not provide much information about which block is on top of which one. One could use the DOM structure and CSS to extract this missing information.

6 Validation Helping Feature

Additionnaly to the identified challenges, we propose three important next features for validation approach that would help solving the challenges: block identification (Section 6.1), traceability (Section 6.2), and comparing the relationship between elements (Section 6.3).

6.1 Block Identification

Currently, our approach is based on the comparison pixel by pixel of two screenshots. Those screenshots are divided into blocks, but those blocks are not considered during the comparison. However, identifying the block in the screenshot would enable one to perform more precise analyses. For example, one can count the number of blocks or compare the pixels of a source block with the rest of the migrated UI.

At the same time, such a feature will ease the traceability feature (see next subsection) and allows one to compute blocks relationship, which is the main concept of what layouts are.

6.2 Traceability

The traceability is the ability to identify blocks couple, *i.e.,* which block in the source application corresponds to which one in the migrated application. We identified two ways to trace blocks: by analyzing the source code of the UI, or by comparing blocks position between source and migrated UI.

For the source code, one can use DOM information to retrieve the block couples. Indeed, DOM elements may have a unique id that can be migrated and

so used to retrieve the block. One can also think of using XPath to retrieve the same element in the UI.

In case the source code of the migrated or the source application does not contain enough information, and if it is not editable, one can rely on comparing blocks position if the block identification (see preceding section) is enabled. Indeed, if the blocks are identified, one can recreate part of the blocks couples by comparing the position of blocks in the source and migrated application. Two blocks with approximately the same position in the source and migrated applications are likely to represent the same UI element.

Having the traceability will allow more precise analyses. Instead of comparing the UI of source application with the migrated one, one will be able to perform the analysis block by block.

6.3 Block Relationship

The block identification should enable the block relationship analysis. Instead of comparing pixel by pixel or block by block, the approach can compare the relationships between the blocks. Indeed, relationships are what define layout.

To do so, we need to extract from source and migrated screenshots the relationship between blocks, and compare them. Such an extraction might be difficult because of the preceding identified challenges. However, dealing with block relationship would be the final step in layout migration validation.

7 Conclusion and Future Work

From a previous experiment, we identified the lack of approach to test the layout of migrated GUI. Moreover, many validation techniques proposed in the literature are getting obsolete with modern frameworks and architectures. In this paper, we explored the state of the art and proposed a new simple approach based on other research fields. Thus, we identified future challenges in layout migration validation.

Finally, we proposed three main future work projects we will study: the block identification in an image, the traceability between source and migrated GUI, and the relationship between the blocks.

Bibliography

[1] Alpuente, M., Romero, D.: A visual technique for web pages comparison. Electronic Notes in Theoretical Computer Science **235**, 3–18 (2009)

[2] Amalfitano, D., Fasolino, A.R., Tramontana, P.: A GUI crawling-based technique for android mobile application testing. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 252–261, IEEE (2011), ISBN 978-1-4577-0019-4, https://doi.org/10.1109/ICSTW.2011.77, URL http://ieeexplore.ieee.org/document/5954416/

[3] Cao, J., Mao, B., Luo, J.: A segmentation method for web page analysis using shrinking and dividing. International Journal of Parallel, Emergent and Distributed Systems **25**(2), 93–104 (2010)

[4] Hayakawa, T., Hasegawa, S., Yoshika, S., Hikita, T.: Maintaining web applications by translating among different RIA technologies. GSTF Journal on Computing p. 7 (2012)

[5] Joorabchi, M.E., Mesbah, A.: Reverse engineering iOS mobile applications. In: 2012 19th Working Conference on Reverse Engineering, pp. 177–186, IEEE (2012), ISBN 978-0-7695-4891-3 978-1-4673-4536-1, https://doi.org/10.1109/WCRE.2012.27, URL http://ieeexplore.ieee.org/document/6385113/

[6] Karami, E., Prasad, S., Shehata, M.: Image matching using sift, surf, brief and orb: performance comparison for distorted images. arXiv preprint arXiv:1710.02726 (2017)

[7] Memon, A., Banerjee, I., Nagarajan, A.: GUI ripping: Reverse engineering of graphical user interfaces for testing. In: Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003), pp. 260–269, IEEE Computer Society Press, Los Alamitos CA (Nov 2003)

[8] Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web **6**(1), 1–30 (2012), ISSN 15591131, https://doi.org/10.1145/2109205.2109208, URL http://dl.acm.org/citation.cfm?doid=2109205.2109208

[9] Moran, K., Watson, C., Hoskins, J., Purnell, G., Poshyvanyk, D.: Detecting and Summarizing GUI Changes in Evolving Mobile Apps. arXiv:1807.09440 [cs] (Jul 2018), URL http://arxiv.org/abs/1807.09440, arXiv: 1807.09440

[10] Morel, J.M., Yu, G.: Asift: A new framework for fully affine invariant image comparison. SIAM journal on imaging sciences **2**(2), 438–469 (2009)

[11] Sánchez Ramón, O., Sánchez Cuadrado, J., García Molina, J.: Model-driven reverse engineering of legacy graphical user interfaces. Automated Software Engineering **21**(2), 147–186 (2014), ISSN 0928-8910, 1573-7535, https://doi.org/10.1007/s10515-013-0130-2, URL http://link.springer.com/10.1007/s10515-013-0130-2

[12] Sánchez Ramón, Ó., Sánchez Cuadrado, J., García Molina, J., Vanderdonckt, J.: A layout inference algorithm for graphical user interfaces. Information and Software Technology **70**, 155–175 (2016)

[13] Sanoja, A., Gançarski, S.: Migrating web archives from html4 to html5: A block-based approach and its evaluation. In: Kirikova, M., Nørvåg, K., Papadopoulos, G.A. (eds.) Advances in Databases and Information Systems, pp. 375–393, Springer International Publishing, Cham (2017), ISBN 978-3-319-66917-5

[14] Van Beusekom, J., Keysers, D., Shafait, F., Breuel, T.M.: Distance measures for layout-based document image retrieval. In: Second International Conference on Document Image Analysis for Libraries (DIAL'06), IEEE (2006)

[15] Verhaeghe, B., Etien, A., Anquetil, N., Seriai, A., Deruelle, L., Ducasse, S., Derras, M.: GUI migration using MDE from GWT to angular 6: An industrial case. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China (2019), URL https://hal.inria.fr/hal-02019015