Automated Repair of Layout Bugs in Web Pages with Linear Programming

Stéphane Jacquet, Xavier Chamberland-Thibeault, and Sylvain Hallé

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

Abstract. The paper addresses the issue of layout bugs, in which elements of a web page may overlap, become misaligned or protrude from their parent container for fortuitous reasons. It proposes a technique to apply corrections to a rendered page by formulating its current state and associated layout constraints into a Mixed Integer Linear Programming problem. An off-the-shelf numerical solver is used to generate a layout that satisfies the constraints, in such a way that disruptions to the original page are minimized. A probe then injects these corrections in the form of a temporary "hotfix". The approach has been implemented and tested on samples of real-world web pages; using techniques that aim to reduce the size of the optimization problem, a solution can often be computed in a few seconds on commodity hardware.

1 Introduction

The complex interaction of HTML, CSS and JavaScript inside a page may cause its elements to be displayed and behave in ways that are not always anticipated by the designer. A recent study of dozens of real-world web sites has shown that bugs related to the user interface of a web page are very frequent, and even occur in high-profile sites such as Facebook or YouTube [7]. Such bugs may have various causes, including cross-browser rendering inconsistencies [14], responsive web design complexities [2, 21] and unforeseen internationalization side effects [15].

Several tools and approaches have been proposed over the past decade to automatically discover such bugs, and potentially identify the elements responsible for the problematic rendering [13, 18, 19, 21]. A page under test may be compared to a reference page, or be evaluated against a set of declarative constraints that it is expected to fulfill. However, much fewer approaches take the problem to the next step, and actually attempt to *repair* the problems that are detected. Web designers are therefore left with the task of finding a suitable fix by themselves. Existing solutions sometimes require many minutes before finding an appropriate fix [8, 14], which makes them unsuitable for on-the-fly corrections.

This is precisely the problem addressed by the present paper, which focuses on a particular class of user interface disruptions called *layout bugs*. These bugs, which are geometrical in nature, occur when the elements of a page are incorrectly placed, have improper dimensions, are misaligned or overlap each other while they should not. Our approach tackles the issue by attempting to generate what we call a "hotfix" —a temporary patch to the properties of elements as they are displayed in the current page, which restores the satisfaction of declarative layout constraints given beforehand. The solution we propose is to convert the state of a page and its constraints into a Mixed Integer Linear Programming problem (MILP). This makes it possible to leverage the use of an industrial-level numerical solver to quickly compute a layout that satisfies the constraints.

This solution faces two key challenges. The first is to keep the size of the numerical model small, in order to produce a result in acceptable time for an end user (seconds rather than minutes). To this end, we introduce the concept of "zone of influence", which allows us to circumscribe in advance the set of elements that may need to be modified in a page, and drastically reduce the number of variables in the corresponding numerical problem. The second challenge is the actual application of the fix into the page; we describe a technique that is guaranteed to impose the given size and position to a given element, which avoids the need to test a candidate repair into an actual browser. We present a proof-of-concept implementation of this technique and report on experimental results; they confirm that our hotfix generation technique can correctly modify the elements of a page to solve a layout bug, in reasonable time for pages of size corresponding to real-world websites.

This paper is structured as follows. Section 2 describes three types of layout bugs with examples, and then covers related works on UI bug detection and repair. Section 3 explains how we handle the correction through a Mixed Integer Linear Program. Section 4 describes a proof-of-concept implementation that has been tested on both real-world and synthetic pages of large size. Section 5 ends the paper with a conclusion, with suggestions of future work.

2 State of the Art for Fixing Layout Bugs in Web Pages

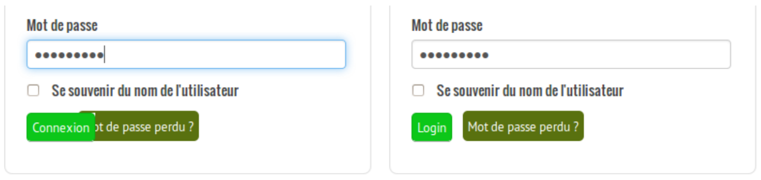
Presentation bugs can routinely be found in web sites, ranging from subtle inconsistencies to more serious errors that may even break a page's functionality. Case in point, a recent study on UI bugs has found more than 100 issues in the pages of various web sites [7]. In this section, we first describe the particular types of layout bugs that are the focus of this work. We then briefly present the various approaches that have been proposed in the past to automatically detect and/or correct such bugs inside web pages.

2.1 Types of Layout Bugs

Among all bugs reported in previous works, we focus in this paper on bugs that are related to a page's geometrical features, namely the size and positioning of the various elements ("boxes") that compose the page; we call these bugs *layout bugs*. In the GUI fault model developed by Lelli *et al.*, these bugs correspond to sub-category GSA1 [12]. Following the terminology introduced by Walsh *et. al.* [21], the layout bugs we consider can be divided in three categories.

Overlapping Elements Figure 1 shows an example of overlapping elements on the login page for an installation of the Moodle[1] platform. The leftmost button, labeled "Connexion", extends over the password recovery button that lies to its right.

[1] https://moodle.org

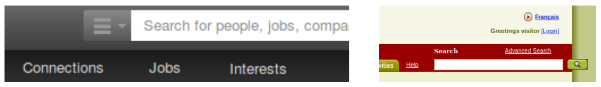


(a) French text (b) English text

Fig. 1: Example of overlapping elements.

The effect produced by overlapping elements is often very easy to spot, yet the causes of the presence of such overlapping elements are multiple. In the example shown above, it appears that the position of the buttons is hard-coded based on the size of the English version of their text. Case in point, Figure 1b shows the effect of changing the button's text to "Login", which restores an eye-pleasing layout.

Misaligned Elements Misaligned elements is a second common type of layout bug, which can sometimes be subtle to detect. An example is shown in Figure 2a, which comes from the LinkedIn platform[2]; in this screenshot, the "Interests" menu is placed one pixel lower than the other elements.



(a) Misalignment (b) Protrusion

Fig. 2: Examples of misaligned and protruding elements.

As with overlapping elements, the causes of misalignment are varied. In the previous example, one can observe by investigating the page's source code that the "Interests" menu is not clickable, contrary to the other elements of the bar. To this end, it is given a different CSS class than the rest of the menu items, which has a slightly different definition for its margins and padding.

Protruding Elements The last type of bug occurs when an element extends beyond the boundaries of another element that should contain it completely. This is what Walsh *et al.* call *element protrusion* [21]. Figure 2b shows an example of such an issue, taken from the site AgentSolo[3]: one can see that the Search button extends beyond the region that is reserved for the menu bar of the page.

[2] `https://linkedin.com`
[3] `https://www.agentsolo.com`

2.2 UI Bug Detection

Over the past decade, several approaches have been proposed for the automated detection of UI bugs in web applications. For example, WebDiff [19] identifies cross-browser layout issues in a web page. The tool harvests the DOM tree of a page on a reference browser multiple times in order to identify the variable elements that should not be considered. Then, it harvests the DOM tree of the same page on other browsers and compares it node by node, to identify mismatches in these nodes' properties and report them.

Walsh *et al.* [21] used Responsive Layout Graphs, which are constructed by querying the DOM tree of a web page under different viewport sizes, to compare a page to itself at various viewport size aiming to find relative layout issues. The DOM trees of each viewport are compared to identify elements that behave incorrectly and report them to the developers. The work by Ryou and Ryu [20] uses a crawler that interacts as much as possible with a web page, building graphs of the page's state along the way. These multiple graphs are then compared: discrepancies between nodes representing the same page element in two graphs are then identified.

All these works are based on the principle of comparison between multiple versions of a page, or between a page and a reference version that is considered correct. An alternate approach consists of asking the developer to write statements that describe the intended appearance of the page in advance. In this line of works, Cornipickle is a web testing tool that focuses on the detection of bugs related to the user interface of a web application [7]. It provides an expressive language in which declarative constraints can be expressed by a web developer, and refer to any visible characteristic of the elements of a page —among others its content, colors, position and dimensions. The tool can then automatically detect violations of these constraints in a web page, and also provides a mechanism for pinpointing the elements of the page that are involved in the corresponding violation.

Similarly, the Cassius framework works as a declarative browser used to verify web pages [18]. Rather than checking constraints on a single rendition of a web page, the tool reasons symbolically over all possible viewports; therefore, if an assertion passes, this guarantees that no possible set of device size or user preferences can ever violate the assertion. In the opposite case, a counterexample layout is produced and an element violating the assertion is identified. Finally, although not a bug detection tool *per se*, we shall mention SeeSS [13], which is a tool that highlights the portions of a web page that are subject to visual modifications when a developer changes a CSS declaration and saves the file that contains it.

2.3 Automated Repair

While most of the aforementioned solutions can identify portions of a page that violate a given condition, none of them attempt to fix the issue. On the contrary, Hallé and Beroual [8] proposed a generic model for correcting abstract objects that do not satisfy a condition, based on the concept of *repair*. Formally, if φ is a condition that some object $o \in O$ violates, and $\tau_1, \dots, \tau_n$ are endomorphisms $O \to O$, a repair is any composition τ of some of the τ_i such that $\tau(o)$ satisfies φ. Intuitively, each τ_i applies a different modification to the object o, and the process of fixing o is reduced to the problem of searching for a combination of modifications that makes the "corrected" object satisfy the condition. Web page layout bugs are cited as one of the potential domains of application of this general theory.

However, the search for an appropriate repair is computationally very expensive, and ultimately amounts to a brute-force generate-and-test algorithm —which, as it turns out, has not been experimentally tested on web pages. Moreover, in this generic model, all potential repairs are seen as black boxes with an equal likelihood of fixing the input object. In the case of violations of layout constraints, which are intrinsically numerical, such a solution does not exploit the geometric nature of the problem to converge more quickly towards a possible fix.

The work closest to the problem we tackle here is an approach to make automated repairs on cross-browser rendering inconsistencies, implemented in a tool called X-Fix [14]. This is done by first comparing an incorrect page with a reference copy, to identify any elements with a different rendering. CSS properties of these elements that have an impact on the rendering discrepancy are identified for each of them. The tool then proceeds to a search for alternate values that could be given to these properties and that would fix the problem, called candidate fixes. The process loops until all bugs have been resolved, or if no improvement on an existing fix can be found.

The generation of candidate fixes is done by implementing a basic numerical solver within the tool, which performs small positive or negative increments to CSS properties of elements and watches for improvements in the value of a fitness score. However, each such candidate requires the page to be re-rendered and re-examined in the browser under test; consequently, experimental results report running times in the order of minutes to find a proper fix. Moreover, in the same way as most tools described in Section 2.2, it requires a correct rendering as a reference. In this respect, one could say that the tool already knows the proper positions and sizes that each element is supposed to have (from the reference page), and searches for appropriate CSS declarations that result in such positions and sizes.

2.4 Optimization-Based Techniques

Several methods using optimization-based techniques have been used in the context of GUIs. A survey has been conducted on using combinatorial optimization for GUI layouts [17]. Mixed Integer Linear Programming (MILP) have been developed in the past two decades to correct layout bugs in web pages. MILP are problems were the objective function and the constraints are described by linear functions. Cassowary [1] was an algorithm developed to solve problems of linear equality and inequality constraints, using a modified version of the simplex algorithm. However, the formulation of the problem initially did not contain an objective function; this means that it was not possible to orient the solver towards preferable solutions.

More recently, the GRIDS system [6] proposed layout management of GUIs using MILP with a multi-objective formulation. A drawback of having multiple functions to optimize is that the user has to chose between a large (theoretically infinite) number of solutions located on a curve called the "Pareto front". To help the user, GRIDS only provides a small sample of feasible pages on that front that are quite different one another. Among the objectives, we find the fact that the outer hull of the GUI is as rectangular as possible, that there are as few holes as possible, and that related elements are grouped together.

The same year, LaaS [11] handled this problem through a MILP while offering two possibilities as objective function. The first one is the selection time. The idea is to make sure that the most important elements take as little time as possible to be found. Typically, the most important items might become bigger than the others elements and closer to the top-left corner. The other one is "visual saliency". It describes basically how the elements catch the eyes of the user. Tests have been done on a single element of attention, which means that a single element of the page needed be found quickly using the time selection criteria or to be catchy using the visual saliency criteria.

3 Modelization of Layout Bugs as MILP

In this section, we describe our proposed approach to fix layout bugs in a web page. Given a page rendered by a browser, and constraints expressed on some of its elements (called alignment, inclusion and containment), violations of these constraints are automatically detected. The page's state and these constraints are converted into a linear optimization problem, which computes new positions and dimensions for the elements; a patch is then directly injected into the page, which restores the layout constraints.

This approach distinguishes itself from existing works in several aspects. First, we do not assume the presence of a reference page, but only declarative constraints that must hold for the specified elements; therefore, our proposed tool must find the proper layout by itself. Second, the goal is to produce a fix on-the-fly, as the user is viewing the page: therefore, the time required to produce a solution should be on the order of a few seconds at most. This is why our approach leverages the use of an external numerical constraint solver, and does not require the re-rendering of the page in order to test candidate fixes. Moreover, the proposed solution relies on a few key techniques that aim to keep the linear optimization problem small.

3.1 Layout Constraints

First, the DOM tree of a web page is modeled as a set of nested rectangles, corresponding to the various HTML elements of the page, from the top-level `<body>` all the way down to individual text leaves (`CDATA`). Each rectangle is defined by the (x, y) coordinates of its top-left corner, its height and its width (in displayed pixels). It follows that a complete web page, made of $n \in \mathbb{N}^*$ elements, is a set of quadruplets $(x^{(i)}, y^{(i)}, w^{(i)}, h^{(i)})$, $i \in \{1, \ldots, n\}$. A web page is described by the characteristics of each rectangle. The complete page is hence a vector of $4n$ components $(x^{(1)}, y^{(1)}, w^{(1)}, h^{(1)}, \ldots, x^{(n)}, y^{(n)}, w^{(n)}, h^{(n)})$.

Essentially, the layout constraints will be expressed on pairs of elements $A = (x^{(a)}, y^{(a)}, w^{(a)}, h^{(a)})$ and $B = (x^{(b)}, y^{(b)}, w^{(b)}, h^{(b)})$; in terms of their position in the DOM tree, these elements will typically be involved either in a parent-child relationship, or a sibling relationship. Each constraint will contribute to the addition of a number of linear equalities or inequalities between some of the variables of the model. We assume that the set of element pairs subject to each type of constraint is given, and known in advance.

Alignment Constraints Alignment constraints are straightforward to handle by linear equalities. For example, if A and B are expected to be aligned vertically, the equality $x^{(a)} = x^{(b)}$ is added to the system. Similarly, the fact that A and B must be aligned horizontally is described by $y^{(a)} = y^{(b)}$. This means that each alignment constraint requires one linear constraint and no extra variables.

Inclusion Constraints The case of inclusion constraints can be handled in a similar way. An element B is completely included within an element A if and only if these four inequalities hold. This means that each inclusion constraint requires four linear constraints and no extra variable.

$$\begin{array}{ll} x^{(a)} \le x^{(b)} & x^{(a)} + w^{(a)} \ge x^{(b)} + w^{(b)} \\ y^{(a)} \le y^{(b)} & y^{(a)} + h^{(a)} \ge y^{(b)} + h^{(b)} \end{array}$$

Disjointedness Constraints It is easy to see that A is disjoint from B if and only if at least one of these four inequalities hold:

$$\begin{array}{ll} x^{(a)} + w^{(a)} \le x^{(b)} & y^{(a)} + h^{(a)} \le y^{(b)} \\ x^{(b)} + w^{(b)} \le x^{(a)} & y^{(b)} + h^{(b)} \le y^{(a)} \end{array}$$

However, such a simple modeling causes problems for linear solvers, which typically cannot directly handle the fact that it suffices that *one* of the constraints must be fulfilled. We therefore propose an alternate modelization, using additional constraints and auxiliary variables. Elements A and B are disjoint if and only if:

$$\begin{array}{rr} x^{(a)} + w^{(a)} \le x^{(b)} + M(1 - z_1) & y^{(b)} + h^{(b)} \le y^{(a)} + M(1 - z_4) \\ x^{(b)} + w^{(b)} \le x^{(a)} + M(1 - z_2) & z_1 + z_2 + z_3 + z_4 \ge 1 \\ y^{(a)} + h^{(a)} \le y^{(b)} + M(1 - z_3) & z_1, z_2, z_3, z_4 \in \{0, 1\} \end{array}$$

where $M \in \mathbb{R}_+$ is a sufficiently large number. Intuitively, the z_i are “choice” variables: setting them to 0 or to 1 determines whether the constraint they are associated with must be fulfilled or not. An equivalent modelization can also be done by replacing the next-to-last equation by $z_1 + z_2 + z_3 + z_4 = 1$. In such a case, z_4 can be removed and replaced by $1 - (z_1 + z_2 + z_3)$, which creates a system with one fewer variable. This means that each disjointedness constraint implies 4 linear constraints and 3 extra binary variables.

Non-decreasing sizes If no constraints on the sizes is given, then some boxes can become smaller. This can lead to some cases where it would be easier for the solver to have a box of length or width equal to 0. Such a thing should not be possible and could happen due to the fact that we lack information in our formulation. In order to avoid that, we add the constraints that the boxes cannot become smaller. Each non-decreasing size constraint adds one linear constraint and no extra variables.

3.2 Defining an Objective Function

Given a set of element pairs that are subject to either alignment, inclusion or disjointedness constraints, it is easy to define a system of inequalities that corresponds to these constraints. Given an input vector $(x^{(1)}, y^{(1)}, w^{(1)}, h^{(1)}, \ldots, x^{(n)}, y^{(n)}, w^{(n)}, h^{(n)})$, a constraint solver will produce an output vector that defines the position and dimensions of each element, such that all constraints are satisfied, if such a solution exists. Therefore, if the original page had a layout that violated one of the constraints, the modifications to the elements' properties describe a way to "fix them".

One could think that merely asking for a solution —any solution— to the solver is sufficient. However, without additional instructions, it is possible that the solver produces a version of the page that satisfies the constraints, but is drastically different from the original. For example, if a single element in a group is misaligned, a valid solution could be to move all elements to a new location in the page. This goes against the intuition that the expected correction would simply move the single misaligned element. Therefore, in order to guide the solver towards solutions that minimally disturb the original document, an *objective function* f must also be provided. A solver can hence be instructed to find solutions that satisfy the constraints, and such that the value of f is minimized.

In the present case, this function should represent the amount of changes made to the original vector. Given an initial page state $(x_0^{(1)}, y_0^{(1)}, w_0^{(1)}, h_0^{(1)}, \ldots, x_0^{(n)}, y_0^{(n)}, w_0^{(n)}, h_0^{(n)})$, the function f to minimize is defined as:

$$\sum_{i=1}^{n} |x^{(i)} - x_0^{(i)}| + |y^{(i)} - y_0^{(i)}| + w^{(i)} - w_0^{(i)} + h^{(i)} - h_0^{(i)}$$

One can see that each term of the sum computes the absolute difference between the initial and the final (x, y) position, and the variation of width and height of each element. Therefore, minimizing f under the layout-bug-free constraints means finding the layout-bug-free web page which is the most similar to the initial web page.

An advantage of this formulation is that the objective function is piecewise linear. Such functions can still be managed through MILP using a proper formulation [5]. One can also note that no absolute values are required for the width and height of the elements, assuming the non-decreasing sizes constraints are used. This allows us to avoid adding $2n$ constraints and $4n$ variables to get the MILP reformulation. Adding those would only lead to longer computation time to solve the MILP.

3.3 Reducing the Number of Constraints

Modeling the previous layout requirements may result in a large number of constraints, affecting an equally large number of elements inside a page. The size of the problem sent to a solver can quickly exceed the limits of what can be handled in reasonable time in terms of user experience. However, the number of variables and constraints can be reduced by taking advantage of the observation that many layout disruptions (and their associated corrections) are local in nature —that is, they have an impact on a limited part of the page, while most of the document typically remains unaffected (and consequently, does not need to be changed).

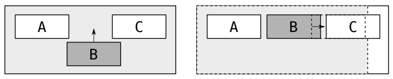


Fig. 3: Displacing an element, and its impact on elements surrounding it.

Let us first consider the case of a correction to an element that requires it to be displaced. For example, Figure 3 shows that element B must be moved up. Doing so without any other change may result in a disjointedness constraint to be violated. Therefore, surrounding elements such as A and C can also have to be moved in order to make room for B at its new location. But this, in turn, can shift elements beyond the original size of their container, and potentially violate a protrusion constraint. In order to fix this issue, the size of the parent element may have to be enlarged to accommodate all elements in their new positions. The end result is the right-hand side of Figure 3; it shows that, when an element needs to be moved, its siblings in the DOM tree may also move, and its parent in the DOM tree may need to be enlarged. A similar reasoning could be made in the case of an element that needs to be enlarged: again, the siblings of this element may need to move, while its parent container may need to be enlarged.

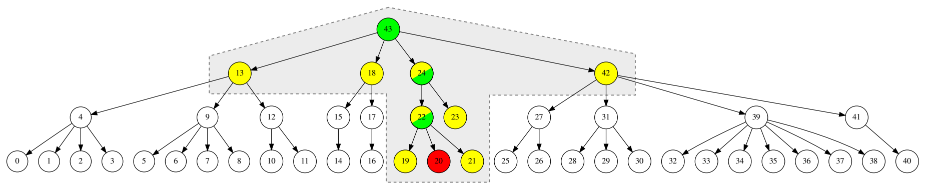


Fig. 4: An illustration of the concept of zone of influence.

These changes cascade recursively through the document: if the parent of an element needs to be resized, then its siblings may need to be moved, and so on. Although this looks like modifications can potentially affect most of the elements, it turns out not to be the case. This is illustrated in Figure 4, which shows an abstract DOM tree. We suppose that node 20, marked in red, needs to be either enlarged or displaced. As per the rules mentioned above, some of the other nodes related to node 20 may need to be displaced (marked in yellow), enlarged (marked in green) or both (marked in both colors). The set of all DOM nodes that are susceptible to either type of correction is called the *zone of influence* of a given element, identified by a gray area in the tree.

Note that a fair fraction of the tree actually stays as is: in our example, 34 of the 44 nodes are assured to require no modification. Consequently, these nodes and their associated constraints do not even need to be included in the model submitted to the solver. This would result, in this case, in a fourfold reduction in the number of variables and constraints.

3.4 Hotfix Application

The schematics of the hotfix generation system can be separated into two phases: the detection phase and the correction phase.

Detection Phase A JavaScript probe P is first injected inside a web page. This probe traverses the DOM tree of the page in a depth-first fashion, and adds to each element a custom attribute called `eid`, whose value is an incrementing number. This procedure makes sure that every element of the page is given a unique numerical identifier. This traversal progressively builds the vector $(x_0^{(1)}, y_0^{(1)}, w_0^{(1)}, h_0^{(1)}, \dots, x_0^{(n)}, y_0^{(n)}, w_0^{(n)}, h_0^{(n)})$, by recording for each element its displayed position and dimensions, as they are rendered by the browser.

This vector is returned to our Java program, which is also provided with a set of constraints C. In its current form, constraints are represented as triplets of the form (i, j, t), where i and j are numerical IDs representing individual elements of the page, and $t \in \{V, H, D, I\}$ indicates whether elements i and j should be respectively vertically aligned, horizontally aligned, disjoint, or if i should be included within j. The program uses these constraints to generate an input model S for a numerical constraint solver. The solution from the solver is a new vector v that stipulates the new position and size of each element in the "corrected" version of the page.

Correction Phase It shall be noted that there is no back-and-forth interaction between the solver and the browser; contrary to some other approaches (notably X-Fix [21]), the candidate solutions examined by the solver do not need to be rendered in real to evaluate their actual effect. This, however, supposes that whatever position and size the solver assigns to elements are guaranteed to be the position and size the elements will indeed have in the corrected page.

It turns out that this task is less trivial than it seems. For example, to place an element e at a particular horizontal coordinate x, it does not suffice to issue a statement such as `e.style.left = x`. The positioning of the element's containing parent, its display properties, as well as additional attributes may lead to the element being placed at a different coordinate than x. In other words, there is a difference between the actual geometry of the element, and the values that must be applied to the element for it to assume the desired geometry.

Changing the width or the height of e is relatively straightforward. The probe starts by measuring the actual rendered properties of the selected element e in the browser, which are retrieved via the `getBoundingClientRect()` method. However, its padding and borders, which are recovered via the `window.getComputedStyle()` method, must first be subtracted from the prescribed dimensions, since both properties are included within the actual width of the element. This newly calculated value is then apply to the element's width or height.

Changing the element's position is more delicate. First, one must determine the reference coordinates of the element, which correspond to the top-left corner of the closest ancestor a whose position is `absolute`, or the document's `<body>` if no such element can be found. Once this ancestor has been found, its absolute positioning and margin sizes are retrieved using the two aforementioned methods. Both of these values must be subtracted from the coordinates returned by the solver, in order to get the position that must actually be applied to the specified element.[4] The `position` attribute of e is then set to `absolute`, and its `top` and `left` properties are set to the calculated values.

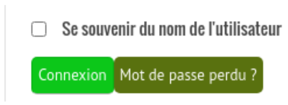
4 Experimental Evaluation

The correction scheme detailed in the previous section has been concretely implemented as a proof-of-concept hotfix generation system for web pages, whose goal is to evaluate the potential of the approach for fixing minor layout bugs. In this section, we describe this implementation and report on experimental results on samples of actual and synthetic web pages.

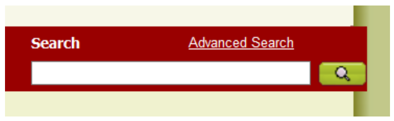
The system is implemented as a Java program, which uses the Selenium WebDriver[5] library to interact with a controllable instance of a web browser. The probe is a custom-made JavaScript piece of code that is injected inside the page by the Java program, and is instructed to extract the properties of elements with specific IDs, for which layout constraints are expected to apply. The Java program then generates an input model for the IBM CPLEX numerical solver [10]. Finally, the solution computed by CPLEX is retrieved by the Java program, and the corresponding JavaScript hotfix is re-injected back into the original page.

4.1 Real-World Bugs

In order to assess the feasibility of the proposed approach to correct actual bugs in a page, two sets of experiments have been conducted, which we describe below. In the first set of experiments, we tested the approach on a sample of real-world web pages presenting layout bugs, taken from a previous study [7]. The goal of these experiments is to determine whether, in an actual web page, the system can not only correctly catch and fix a layout constraint stipulated by the designer beforehand, but also avoid disturbing other (correct) parts of the page. In each of the sampled pages, the appropriate constraints (alignment, containment or disjointedness) on the faulty element and its neighbors were manually identified. The page, along with these constraints, was then sent to our hotfix generator, and the result of the hotfix application was then visually inspected.



(a) Moodle



(b) AgentSolo

Fig. 5: Examples of the application of a hotfix to an incorrect page.

[4] Except if p is the document's `<body>`, which behaves differently and where margins must be ignored.

[5] `https://seleniumhq.org`

Figure 5a shows an example of the application of a hotfix, on the Moodle page already shown in Figure 1a. One can observe that the two green buttons, which overlapped in the original page, are now placed exactly side by side. Although no constraint was expressed on the alignment of these two buttons, they remain horizontally aligned. This is due to the fact that the numerical solver is instructed to minimize the changes applied to the original document: moving any of the two buttons up or down would amount to a greater total change to the page than simply keeping them aligned. This also explains why the size of each button has been left unchanged.

We obtained similar results for the other pages we tested. For example, Figure 5b shows the hotfix for the protruding button illustrated in Figure 2b. Note how the red menu bar has been extended in width exactly enough to contain the search button. Although not visible in this screenshot, the rest of the page remained untouched.

4.2 Synthetic Pages

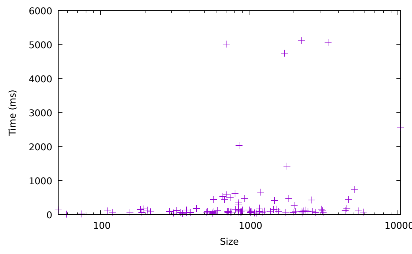
MILPs formulations are easy to formulate but they are NP-hard, implying that their worst case complexity is not polynomial [16]. Therefore, a second set of experiments aims to measure the scalability of the approach with respect to the number of elements in a page. To this end, we conducted a systematic stress test by running our hotfix tool on a sample of synthetic DOM trees, produced by a random page generator we implemented and called PageGen[6]. The experiments have been implemented in the form of a LabPal package [9] that is publicly available online[7].

The page generator works recursively as follows. First, for a given element p, a number of children c is randomly selected. For each of these children, a depth d is also randomly selected. If $d = 0$, an element b, of randomly selected width and height, is created and added as a child of p. Otherwise, b is recursively populated before being added to p's children. Once all the children of p have been created, the last step is to arrange them inside p, either following a horizontal or a vertical layout, separated by an equal margin. Once the elements are arranged, the dimensions of p are set to a rectangle that includes all the children. The end result is a tree of nested rectangles, which can be exported as an HTML document made of `<div>` elements, one for each box. Since the goal of our approach is to correct properties related to the position and size of arbitrary elements, the actual tag names and their content are irrelevant.

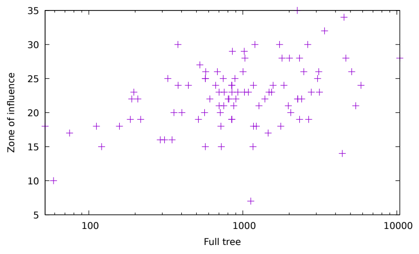
What is more, layout bugs can also be artificially injected when a page is generated. Once all elements are arranged within their parent (according to the horizontal or vertical layout), a coin is flipped for each to determine whether the element should be purposefully misaligned with respect to the others, moved to overlap with one of its siblings, or enlarged to extend beyond the dimensions of its parent. Using this generator, we produced a sample of 100 generated web pages, which includes trees ranging between 2 and 10450 elements. We shall mention that a recent empirical analysis of real-world web sites observed that 90% of pages had fewer than 2,000 nodes [4]. Therefore, we can safely conclude that our sample contains pages of size comparable to (and even larger than) sites that are actually present on the web.

[6] `https://github.com/sylvainhalle/pagegen`
[7] `https://github.com/liflab/hotfix-lab`

For each of these pages, we measured the total time required to generate and apply a hotfix to the layout bug contained in the page. Since the goal is to generate fixes on-the-fly, the solver was given a very short time budget to produce a result (at most 2 seconds). The running times are shown in Figure 6a, plotted in function of page size. One can see that, for most pages, solving time remains well under 1 second; only a few pages exceeded the timeout. These running times should be compared with those obtained by X-Fix [14], which reports a median solving time of 841 seconds on a sample of web sites containing an average of 425 DOM nodes each. Our faster running times, however, are crucial, since our goal is to produce a fix to a page on-the-fly, and not to find a more permanent way of correcting the issue at the CSS level.



(a) Running time



(b) Impact of zone of influence

Fig. 6: Experimental results from our benchmark.

Figure 6b is a visualization of the impact that the use of zones of influence has on the analysis of a page. Each point in this plot represents a pair of trees: on the x-axis is the original size of the tree, and on the y-axis the size of the tree trimmed to retain only the zone of influence of the faulty elements. One can see the drastic reduction in the number of nodes that need to be modeled: pages of thousands of DOM nodes are reduced to a portion containing a few dozens at most. Without such a reduction, the MILP problem to solve would quickly become intractable.

5 Conclusion and Future Work

In this work, we proposed a technique for automatically generating repairs in the case where a web page violates conditions on the layout of its DOM elements. The problem has been modeled as a MILP problem, using an objective function that aims to minimize the disruptions introduced into the page to restore the conditions. The approach has been implemented as a proof-of-concept tool using a combination of the Selenium browser driver for page manipulation, and the IBM CPLEX software for solving numerical constraints. An experimental evaluation of this implementation has shown that our hotfix generation technique can correctly modify the elements of a page to solve a layout bug (§4.1); moreover, the introduction of the concept of zone of influence can reduce the optimization problem and produce results in reasonable time in terms of user experience (a few seconds), for pages of size corresponding to real-world websites (§4.2).

However, these conclusions rest on several hypotheses, which we discuss below. First, the proposed approach shares an issue that is common to all declarative systems based on assertions: in order for bugs to be detected and pages to be fixed, a page must be accompanied by appropriate constraints that should be followed by its elements. Moreover, these constraints must be *complete*, in the sense that any page that satisfies them should be considered valid. It turns out that the human eye makes many implicit assumptions over the expected size, position and alignment of elements which, in our approach, must be explicitly provided. For example, without further constraint, the fix shown in Figure 5b will correctly enlarge the red menu area, but not the white parent element that contains this menu. To decrease the burden on the designer of writing such tedious conditions, a future work we consider is to automatically deduce such "obvious" layout conditions based on heuristics.

The hotfix generated by our approach may only modify the page in a subtle way visually, however it alters the structure of the page in a drastic way. Each element to which a patch is applied has its position property set to "absolute": this makes sure that changing its `top` and `left` attributes are guaranteed to move the element to the exact location stipulated by the solver. This avoids having to test a candidate fix to make sure it has the intended effect, as needs to be done in tools such as X-Fix. However, resizing the page after the application of the hotfix may result in the element being yet again incorrectly placed; an immediate workaround is to recompute a new hotfix when this happens. Absolute positioning also removes the element from the normal flow inside its parent container. For an element that is relatively positioned, an alternate fix, which involves modifying the element's margins, is currently being worked on.

We have also seen that the number of variables and constraints sent to the solver was kept to a manageable level thanks to the observation that only elements in a so-called "zone of influence" need to be modeled. However, this only works under the hypothesis that no element is ever reduced in size, because changes only propagate upwards through the DOM tree. In contrast, if an element can be made smaller, then this change propagates *downwards* to all its children, and in such a case, the zone of influence of an element becomes the whole document. Circumscribing the zone of influence in the case of element reductions is the subject of ongoing work, which shall be integrated in a future version of our system.

Finally, we plan to integrate this hotfix generation mechanism directly into the Cornipickle declarative testing tool, and extend it to other types of constraints beyond the three types of layout bugs addressed in this paper. Another step would be to test the tool on a larger sample of bugs and websites, and reenact the same tests on different browsers to ensure the complete validity of the tool, since all the present tests have been realized only on Chrome.

References

1. Badros, G.J., Borning, A., Stuckey, P.J.: The Cassowary linear arithmetic constraint solving algorithm. ACM Trans. Comput.-Hum. Interact. **8**(4), 267–306 (2001)

2. Beroual, O., Guérin, F., Hallé, S.: Detecting responsive web design bugs with declarative specifications. In: Bieliková et al. [3], pp. 3–18
3. Bieliková, M., Mikkonen, T., Pautasso, C. (eds.): ICWE, Lecture Notes in Computer Science, vol. 12128. Springer (2020)
4. Chamberland-Thibeault, X., Hallé, S.: Structural profiling of web sites in the wild. In: Bieliková et al. [3], pp. 27–34
5. Croxton, K., Gendron, B., Magnanti, T.: A comparison of mixed-integer programming models for non-convex piecewise linear cost minimization problems. Management Science **49**, 1268–1273 (2003)
6. Dayama, N.R., Todi, K., Saarelainen, T., Oulasvirta, A.: GRIDS: Interactive layout design with integer programming. In: CHI. pp. 1–13. ACM (2020)
7. Hallé, S., Bergeron, N., Guerin, F., Breton, G.L., Beroual, O.: Declarative layout constraints for testing web applications. J. Log. Algebraic Methods Program. **85**(5), 737–758 (2016)
8. Hallé, S., Beroual, O.: Fault localization in web applications via model finding. In: Gößler, G., Sokolsky, O. (eds.) CREST@ETAPS. EPTCS, vol. 224, pp. 55–67 (2016)
9. Hallé, S., Khoury, R., Awesso, M.: Streamlining the inclusion of computer experiments in a research paper. Computer **51**(11), 78–89 (2018)
10. IBM: IBM ILOG CPLEX optimization studio CPLEX user's manual, version 12 release 6 (2013), `https://public.dhe.ibm.com/software/products/Decision_Optimization/docs/pdf/usrcplex.pdf`
11. Laine, M., Nakajima, A., Dayama, N.R., Oulasvirta, A.: Layout as a service (LaaS): A service platform for self-optimizing web layouts. In: Bieliková et al. [3], pp. 19–26
12. Lelli, V., Blouin, A., Baudry, B.: Classifying and qualifying GUI defects. In: ICST. pp. 1–10. IEEE Computer Society (2015)
13. Liang, H., Kuo, K., Lee, P., Chan, Y., Lin, Y., Chen, M.Y.: SeeSS: seeing what I broke - visualizing change impact of cascading style sheets (CSS). In: Izadi, S., Quigley, A.J., Poupyrev, I., Igarashi, T. (eds.) UIST. pp. 353–356. ACM (2013)
14. Mahajan, S., Alameer, A., McMinn, P., Halfond, W.G.J.: Automated repair of layout cross browser issues using search-based techniques. In: ISSTA. pp. 249–260. ISSTA 2017, ACM (2017)
15. Mahajan, S., Alameer, A., McMinn, P., Halfond, W.G.J.: Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques. In: ICST. pp. 215–226. IEEE Computer Society (2018)
16. Morrison, D.R., Jacobson, S.H., Sauppe, J.J., Sewell, E.C.: Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization **19**, 79–102 (2016)
17. Oulasvirta, A., Dayama, N., Shiripour, M., John, M., Karrenbauer, A.: Combinatorial optimization of graphical user interface designs. Proceedings of the IEEE **PP**, 1–31 (2020)
18. Panchekha, P., Torlak, E.: Automated reasoning for web page layout. In: Visser, E., Smaragdakis, Y. (eds.) OOPSLA. pp. 181–194. ACM (2016)
19. Roy Choudhary, S., Versee, H., Orso, A.: WEBDIFF: Automated identification of cross-browser issues in web applications. In: ICSM. pp. 1–10. IEEE (2010)
20. Ryou, Y., Ryu, S.: Automatic Detection of Visibility Faults by Layout Changes in HTML5 Web Pages. In: ICST. pp. 182–192. IEEE (2018)
21. Walsh, T.A., Kapfhammer, G.M., McMinn, P.: Automated layout failure detection for responsive web pages without an explicit oracle. In: Bultan, T., Sen, K. (eds.) ISSTA. pp. 192–202. ACM (2017)