Verifying that Web Pages have Accessible Layout

Pavel Panchekha pavpan@cs.uw.edu
Adam T. Geller atgeller@uw.edu
Michael D. Ernst mernst@cs.uw.edu
Zachary Tatlock ztatlock@cs.uw.edu
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle, WA, USA

Shoaib Kamil kamil@adobe.com
Creative Intelligence Lab
Adobe Research, NY, USA

Abstract

Usability and accessibility guidelines aim to make graphical user interfaces accessible to all users, by, say, requiring that text is sufficiently large, interactive controls are visible, and heading size corresponds to importance. These guidelines must hold on the infinitely many possible renderings of a web page generated by differing screen sizes, fonts, and other user preferences. Today, these guidelines are tested by manual inspection of a few renderings, because 1) the guidelines are not expressed in a formal language, 2) the semantics of browser rendering are not well understood, and 3) no tools exist to check all possible renderings of a web page. VizAssert solves these problems. First, it introduces *visual logic* to precisely specify accessibility properties. Second, it formalizes a large fragment of the *browser rendering algorithm* using novel *finitization reductions*. Third, it provides a *sound, automated tool* for verifying assertions in visual logic.

We encoded 14 assertions drawn from best-practice accessibility and mobile-usability guidelines in visual logic. VizAssert checked them on on 62 professionally designed web pages. It found 64 distinct errors in the web pages, while reporting only 13 false positive warnings.

Keywords accessibility, usability, verification, SMT, layout, CSS, semantics

ACM Reference Format:
Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying that Web Pages have Accessible Layout. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3192366.3192407

1 Introduction

Web page accessibility is imperative for moral and legal reasons. There are over 7.4 million adults with a vision impairment in the USA alone [35]. Other users have sensorimotor disabilities that restrict their use of input devices. These people should have the opportunity to use web pages just as other people do, but inaccessible web pages lock them out. GUI designers strive to make their pages accessible by following guidelines, such as those from the Department of Justice [50] or the W3C's Web Content Accessibility Guidelines [53]. For some applications, adherence to these guidelines is mandated by the European Web and Mobile Accessibility Directive [16], the Americans with Disabilities Act [49], or the Basic IT Law of Japan [15]; other countries also have such standards. In 2017, at least 814 lawsuits were brought against web site owners for inaccessible web pages [52].

Usability guidelines are similar: from an accessibility perspective a developer may want to ensure that buttons are large enough for users with sensorimotor disabilities, whereas from a usability perspective a developer may want to ensure that buttons are large enough for users on a mobile device.

Accessibility and usability guidelines must be satisfied on all of the infinitely many combinations of *rendering parameters* such as screen size, default font sizes, and user preferences. Currently, developers test for adherence to these guidelines by running the application with a few chosen parameters and visually searching for violations of design constraints, or they use automated pixel-by-pixel comparison engines such as Selenium, Sikuli, and Raven [9, 13, 17]. These techniques only consider a particular subset of rendering parameters and are not sufficient to ensure that applications are accessible [24, 29, 43, 46]. To verify accessibility requires checking all possible combinations of rendering parameters.

We introduce VizAssert, a tool for verifying that an assertion about visual layout holds for web pages. Web pages are a ubiquitous type of user interface for modern applications. VizAssert takes as input a web page and an assertion in a formal *visual logic*, for example a minimum font size or the visibility and contrast of widgets. VizAssert either soundly certifies that the assertion is satisfied for all rendering parameters in a user-chosen bounded set, or it provides a counterexample for which the assertion is violated.

VizAssert reasons about web page layout using a formalization of the browser rendering algorithm specified by the W3C CSS 2.1 standard [54]. The CSS standard describes several key features described in terms of operations on arbitrary-sized sets, which are difficult for an SMT solver to reason about. This prevented previous attempts from formalizing several ubiquitous aspects of CSS, among them line height computation, margin collapsing, and floating layout [39]. Our work introduces novel, sound reductions (*finitzation reductions*) from the arbitrary-size sets discussed by the standard to a compact, finite-size representation which is more amenable to mechanical reasoning. By

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
2018. ACM ISBN 978-1-4503-5698-5/18/06...$15.00
https://doi.org/10.1145/3192366.3192407

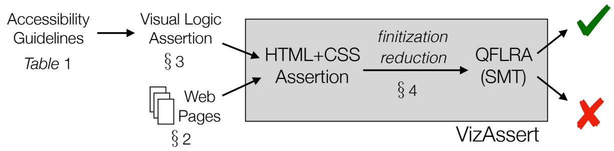


Figure 1. VizAssert ensures that web pages satisfy accessibility guidelines. VizAssert transforms those assertions, expressed in visual logic, into properties of the HTML and CSS source of the web page, then encodes those properties to formulas in the quantifier-free theory of linear real arithmetic.

reducing arbitrary-sized sets to finite analogs, VizAssert encodes its formalization of browser rendering to an efficiently-solvable SMT query and uses Z3 [14] to automatically search for counterexamples to an assertion or to certify that no counterexamples exist.

1.1 VizAssert's Input and Output

Web developers wish to ensure that their web pages satisfy high-level, English-language guidelines. Developers are eager to fix accessibility issues, but discovering those issues is too difficult [29, 46]. The developer provides to VizAssert a web page and a guideline, expressed as a formal assertion in visual logic.

Figure 1 shows how VizAssert works. Using its formalization of the browser rendering algorithm, VizAssert constructs a search problem equivalent to the truth of that assertion. An answer to the search problem is a set of rendering parameters (that is, a set of browser preferences) such that the rendering violates the assertion. VizAssert encodes this search problem into a formula in the quantifier-free theory of linear real arithmetic (QFLRA), and invokes an SMT solver to decide the search problem.

VizAssert's output is "verified" if no rendering parameters can be found that cause a violation of the assertion. The assertion provably holds for all considered rendering parameters. VizAssert's output is "violated" if the search problem is solvable. VizAssert also outputs *counterexample* that identifies a set of boxes on the page and values for the rendering parameters (window size, font size) for which the assertion is violated. The developer can examine the counterexample in a browser to confirm the error and then modify the page to fix it.

The errors VizAssert detects are not crashes or exceptions. The lack of notifications makes developers more likely to overlook accessibility failures. This makes verification tools like VizAssert especially valuable.

#	Description	Source
General assertions, applicable to many web pages		
1	Text is at least 14px tall	[53]
2	Text can be resized by up to 200%	[53]
3	Lines are no more than 80 characters	[53]
4	Elements for screen-reader users are off-screen	[34, 41]
5	The page does not require horizontally scrolling	[8]
6	Headings form a visual hierarchy	[38]
7	Text does not overlap	[2]
8	Lines are appropriately spaced	[7]
Specific assertions, applicable to a single web page		
9	Text has sufficient contrast	[1, 53]
10	Text does not overlap image	[1, 53]
11	Dropdown menus are hidden when not selected	[56]
12	Columns are vertically aligned	[7]
13	Full link text is visible	[2]
14	Main button is big enough	[48]

Table 1. Best-practice accessibility and usability guidelines that we formalized in visual logic and evaluated on professionally designed web pages (Section 5). Appendix B formalizes each guideline.

1.2 Case Studies

This paper considers 14 accessibility guidelines and best practices (Table 1). They all represent real concerns for designers, but the particular set is less important than the fact that VizAssert is general and extensible.

We formalized each accessibility guideline in visual logic and checked them on a collection of 62 professionally designed web pages. Section 3 discusses the adaptations and choices necessary to turn informal guidelines and best practices into formal assertions. 6 of the assertions are page-specific, demonstrating that users can write custom assertions to verify domain-specific properties of their web pages. In all, there are 502 combinations of web page and assertion. Of these, VizAssert found 64 assertion violations, and issued 13 false positive warnings.

1.3 Contributions

In summary, this paper's contributions are:

- *Visual logic*, a language to express visual layout properties (Section 3), including the 14 accessibility and usability guidelines in Table 1.
- An efficient formalization of many crucial components of the browser rendering algorithm. For space reasons, this paper highlights three of them—line height (Section 4.1), margin collapsing (Section 4.2), and floating layout (Section 4.3)—that utilize a key implementation tactic (finitization reductions).

- An open-source implementation, VizAssert (https://cassius.uwplse.org/), that automatically verifies a visual assertion for a web page, or produces a counterexample user configuration for which the assertion is violated.
- An evaluation of VizAssert on 62 professionally designed web pages, identifying 64 errors with only 13 false positives (Section 5).

2 Background

This section describes the W3C browser rendering algorithm [54], which turns HTML and CSS into a tree of boxes annotated with colors, sizes, and positions. VizAssert's formalization of a subset of this rendering algorithm is based on and significantly extends that of Cassius [39]. The most noteworthy extensions are described in Section 4.

HTML HTML defines *elements* and *text*. Each element has a *tag name*, and text is placed within and between elements. For example, the following HTML represents 4 elements (with tags html, body, b, and button) and four pieces of text:

```
<html><body>This is <b>formatted</b> text
and a <button>button</button></body></html>
```

Some HTML elements, like the button element, are specially interpreted by the browser and rendered with browser- or OS-specific methods. Most other elements have no special behavior: browsers provide a default CSS file to ensure that, for example, text inside a b-tagged element is bold. It is this default CSS file, not something intrinsic to the b tag, that causes the text to render in bold.

CSS A CSS file defines a sequence of *rules*. Each rule *selects* certain elements and then sets the values of various *properties* on those elements. The following CSS file contains two rules:

```
body { margin: .5em; }
b { font-weight: bold; }
```

The first rule selects elements with the body tag and sets their margin property. The second rule selects elements with the b tag and sets their font-weight property. CSS also allows selecting elements by their relationship to other elements or by attributes attached to those elements. If two applicable rules conflict, the rule with the most specific selector is used.

Rendering The browser combines the HTML and CSS to produce a rendering: a *box tree* containing all of the page's visual elements. The browser uses the *rendering parameters*, including the height and width of the browser window and the user's preferred font size, as inputs to this rendering process. Numerous browser subsystems interact to render HTML and CSS (Figure 2):

- *Selectors* are matched to elements to find the values specified for every CSS property for every element.
- *Cascading* determines which rule's value to use when multiple rules apply to one element.

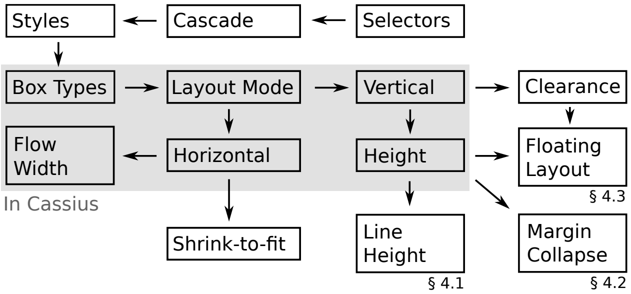


Figure 2. The major subsystems of VizAssert's formalization of web page layout. Each component is described briefly in Section 2. Many details, such as VizAssert's handling of color, media queries, and width and height bounds, are hidden in this picture, which shows only the highest-level components. Components outside the shaded area are newly formalized in this work.

- *Styles* are computed for each element, resolving references and relative values in the specified CSS values.
- *Box types* are determined for each box using the computed value of the display property.
- *Layout modes* are selected for each box using the computed values of the display, float, and position properties, plus the box type.
- *Horizontal* widths and positions are computed for boxes. This computation is different for different layout modes.
- *Flow widths* are used for most boxes, computed from their width value and the width of their containing box.
- *Shrink-to-fit* widths are used instead for boxes whose width value is auto and which use certain layout modes.
- *Vertical* positions are computed for boxes based on the positions of previous boxes and, in some cases, the positions of floats.
- *Heights* are computed for most boxes based on the positions and sizes of their contents.
- *Line heights*, for lines of text, are computed separately, using information about the size of fonts and details of text layout described in Section 4.1.
- *Margins* sometimes *collapse*, allowing adjacent vertical margins of boxes to overlap, following rules discussed in Section 4.2.
- *Clearance* is determined for elements whose clear property is set. Clearance moves those elements so that they do not have a floating box beside them.
- *Floating layout* allows elements to move to the right or left of their parent and have text wrap around them. Section 4.3 describes this subsystem.

Once the size and position of every box is known (as well as colors and other miscellaneous properties), the web page can be drawn to the screen.

$$\langle \mathit{assertion} \rangle ::= \forall b_1, \ldots \in \mathcal{B} : \langle \mathit{cond} \rangle$$

$$\begin{aligned}\langle \mathit{cond} \rangle ::= {} & \langle \mathit{cond} \rangle \wedge \langle \mathit{cond} \rangle \mid \neg \langle \mathit{cond} \rangle \mid \langle \mathit{cond} \rangle \vee \langle \mathit{cond} \rangle \\ \mid {} & \langle \mathit{real} \rangle = \langle \mathit{real} \rangle \mid \langle \mathit{real} \rangle < \langle \mathit{real} \rangle \mid \langle \mathit{real} \rangle > \langle \mathit{real} \rangle \\ \mid {} & \langle \mathit{box} \rangle = \langle \mathit{box} \rangle \mid \langle \mathit{box} \rangle.\mathsf{type} = \langle \mathit{type} \rangle \mid \langle \mathit{box} \rangle.\mathsf{whitespace}\end{aligned}$$

$$\begin{aligned}\langle \mathit{real} \rangle ::= {} & \mathbb{R} \mid \langle \mathit{real} \rangle + \langle \mathit{real} \rangle \mid \langle \mathit{real} \rangle - \langle \mathit{real} \rangle \mid \mathbb{R} \times \langle \mathit{real} \rangle \\ \mid {} & \langle \mathit{box} \rangle.\langle \mathit{dir} \rangle \mid \langle \mathit{color} \rangle.\mathsf{r} \mid \langle \mathit{color} \rangle.\mathsf{g} \mid \langle \mathit{color} \rangle.\mathsf{b}\end{aligned}$$

$$\begin{aligned}\langle \mathit{color} \rangle ::= {} & \mathsf{transparent} \mid \mathsf{rgb}(\langle \mathit{real} \rangle, \langle \mathit{real} \rangle, \langle \mathit{real} \rangle) \\ \mid {} & \langle \mathit{box} \rangle.\mathsf{fg} \mid \langle \mathit{box} \rangle.\mathsf{bg} \mid \gamma(\langle \mathit{box} \rangle.\mathsf{fg}) \mid \gamma(\langle \mathit{box} \rangle.\mathsf{bg})\end{aligned}$$

$$\begin{aligned}\langle \mathit{box} \rangle ::= {} & b_i \mid \mathsf{root} \mid \mathsf{null} \mid \langle \mathit{box} \rangle.\mathsf{ancestor}(\langle \mathit{cond}^* \rangle) \\ \mid {} & \langle \mathit{box} \rangle.\mathsf{parent} \mid \langle \mathit{box} \rangle.\mathsf{first\text{-}child} \mid \langle \mathit{box} \rangle.\mathsf{last\text{-}child} \\ \mid {} & \langle \mathit{box} \rangle.\mathsf{next} \mid \langle \mathit{box} \rangle.\mathsf{prev}\end{aligned}$$

$$\langle \mathit{type} \rangle ::= \mathsf{window} \mid \mathsf{inline} \mid \mathsf{line} \mid \mathsf{text} \mid \mathsf{block}$$

$$\langle \mathit{dir} \rangle ::= \mathsf{top} \mid \mathsf{right} \mid \mathsf{bottom} \mid \mathsf{left}$$

Figure 3. Visual logic, a language for formally describing visual layout properties. All quantifiers are over the set $\mathcal{B}$ of boxes in a rendering.

3 Visual Logic

Visual logic expresses accessibility and usability assertions in formal and unambiguous terms. This is the input to VizAssert.

3.1 Core Visual Logic

Visual logic expresses properties of a visual layout. Visual logic allows quantified formulas over a simple expression language with conditional values, real numbers, boxes, and colors (Figure 3). These formulas describe properties of a tree of rectangular boxes.

Most visual logic constructs are straightforward (and described in more detail in Appendix A). Some domain-specific constructs include:

- Assertions are universally quantified over boxes "b_i".
- "$\langle \mathit{box} \rangle.\mathsf{type} = \langle \mathit{type} \rangle$" checks the type of a box, which defines the sort of content the box contains.
- "$\langle \mathit{box} \rangle.\mathsf{whitespace}$" determines whether a text box contains only whitespace or also contains other text.
- The "$\langle \mathit{box} \rangle.\langle \mathit{dir} \rangle$" construct expresses the position of the box's edges (and thus the box's position and size).
- The γ function gamma-corrects RGB colors.
- A box's siblings, children, and parent are accessible through its previous, next, parent, first-child, and last-child fields, which can have a null value.

Visual logic makes several design choices to enable efficient reasoning with an SMT solver. We did not find any assertions that we were unable to formalize due to these limitations. (1) Since SMT solvers can only efficiently reason about *linear* real arithmetic, multiplications must involve a constant. (2) Visual logic allows only universally-quantified queries, because these correspond to quantifier-free queries to the SMT solver. (3) SMT solvers cannot reason about recursive properties, so visual logic provides the ancestor function, which represents the closest ancestor of a box that satisfies a conditional expression in one variable (written "?"). If no ancestor satisfies the ancestor condition, the null box is returned. For example, "$b.\mathsf{ancestor}(?.\mathsf{bg} \neq \mathsf{transparent}).\mathsf{bg}$" refers to the background color of the nearest ancestor of b whose background isn't transparent. Formally,

$$\begin{aligned}&\mathsf{null}.\mathsf{ancestor}(C) = \mathsf{null} \\ &b.\mathsf{ancestor}(C) = b \textbf{ if } C[?/b] \textbf{ else } b.\mathsf{parent}.\mathsf{ancestor}(C)\end{aligned}$$

Evaluating a visual logic assertion on a concrete rendering of a GUI is straightforward. Searching the infinite set of all possible renderings for a counterexample to the assertion is more challenging (see Section 4).

3.2 Visual Logic for the Web

Visual logic formalizes properties of visual layouts for any platform that structures graphical interfaces as a tree, such as HTML and CSS, Swing [19], Cocoa [3], or Android [42]. Adapting visual logic to one of these platforms involves adding platform-specific constructs to visual logic. Since this paper focuses on the accessibility of web pages, it specializes visual logic for the web platform by adding support for selectors and the CSS box model.

Selectors Visual logic for the web contains a conditional expression to determine whether a box is generated by an element that matches a given CSS selector [54]:

$$\langle \mathit{cond} \rangle ::= \cdots \mid \langle \mathit{box} \rangle \in \$(\langle \mathit{selector} \rangle)$$

For example, "$b \in \$(\mathsf{h1}, \mathsf{h2})$" evaluates to true if b is the box of a first- or second-level heading. Adaptations of visual logic to other platforms could add analogous concepts, such as access to Swing component names, Cocoa viewIDs, or android:ids on Android.

Box model Visual logic for the web supports references to the position of the content, padding, border, and margin edges of a box, with the border edge as the default.

$$\langle \mathit{real} \rangle ::= \cdots \mid \langle \mathit{box} \rangle.\langle \mathit{dir} \rangle[\langle \mathit{edge} \rangle]$$

$$\langle \mathit{edge} \rangle ::= \mathsf{margin} \mid \mathsf{border} \mid \mathsf{padding} \mid \mathsf{content}$$

For example, the conditional expression "$b_1.\mathsf{top}[\mathsf{margin}] = b_2.\mathsf{top}[\mathsf{margin}]$" evaluates to true if b_1's and b_2's margin borders are aligned at the top.

3.3 Formalizing Assertions

Two broad challenges must be overcome to translate informal, English-language guidelines to assertions in visual logic: precisely identifying the page elements discussed by the guideline and translating high-level design concepts to concrete visual properties. To illustrate each, we use a representative assertion from the list in Table 1. Appendix B presents the formalizations of all 14 assertions.

Identifying relevant elements Users who are partially or fully blind often navigate the web using a screen-reader. Content intended solely for such users is often positioned off-screen, where sighted users will not see it. Assertion #4, "elements for screen-reader users are off-screen" requires that these elements stay off-screen.

$$\forall b \in \mathcal{B} : \mathsf{for\_screenreader}(b) \implies \neg\mathsf{onscreen}(b)$$

The screen is represented by the root box:

$$\mathsf{onscreen}(b) := b.\mathsf{right} \geq \mathsf{root.left} \wedge b.\mathsf{bottom} \geq \mathsf{root.top}$$

The majority (38) of the 62 evaluation pages featured social sharing buttons[1] implemented by setting those buttons' background-image property to those social networks' logos. Since background images are not accessible to blind users, these pages also included text naming the service, but positioned this text off-screen.

$$\mathsf{for\_screenreader}(b) := \\ b.\mathsf{type} = \mathsf{text} \wedge \mathsf{is\_descendant}(b, \#S)$$

where S names a social service. The is_descendant function is a variant of ancestor:

$$\mathsf{is\_descendant}(b, S) := b.\mathsf{ancestor}(? \in \$(S)) \neq \mathsf{null}$$

Concretizing design concepts Web designers use visual details, like size or spacing, to suggest to sighted users which content on the page is more or less important. Users who browse the web with a screen-reader instead read the hierarchical heading tags h1–h6, where smaller numbers describe more important content. However, web page developers can change the appearance of these tags, making screen-reader users and sighted users perceive a different hierarchy of importance in page content. Assertion #6, "Headings must form a visual hierarchy", checks that no such misuse occurs.

$$\forall b_1, b_2 \in \mathcal{B} : \mathsf{in\_header}(b_1) \wedge \mathsf{in\_header}(b_2) \wedge \\ \mathsf{header\_level}(b_1) < \mathsf{header\_level}(b_2) \implies \\ \mathsf{visual\_importance}(b_1) > \mathsf{visual\_importance}(b_2)$$

We used text height to establish visual importance on the evaluation pages: visual_importance(b) := b.height. Selecting headings is done using is_descendant:

$$\mathsf{in\_header}(b) := b.\mathsf{type} = \mathsf{text} \wedge \neg b.\mathsf{whitespace} \wedge \\ (\mathsf{is\_descendant}(b, \mathsf{h1}) \vee \mathsf{descendant}(b, \mathsf{h2}) \vee \cdots)$$

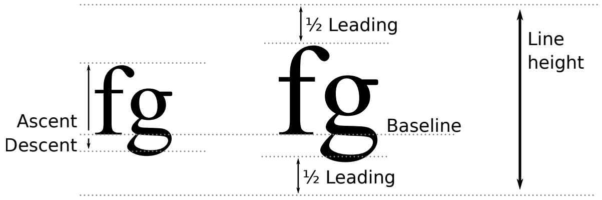


Figure 4. Line height is computed by aligning all text in the line to a common baseline and finding the maximum ascents above and descents below the baseline, including blank space above and below the text called leading.

Once a heading is selected, its level can be computed from its tag name:[2]

$$\mathsf{header\_level}(b) := \textbf{if } \mathsf{is\_descendant}(b, \mathsf{h1}) \textbf{ then } 1 \textbf{ else } \ldots$$

Though identifying relevant elements and concretizing design concepts can be challenging is some cases (see Appendix B), visual logic provides an expressive language for formalizing accessibility and usability guidelines.

4 Formalization of Browser Rendering

VizAssert uses a formalization of the W3C browser rendering algorithm to determine all possible renderings of a web page, given its source code (CSS & HTML). This formalization builds upon earlier work and contains many advancements to support realistic web pages (see Section 7.1). This section describes a central aspect of these advances: a novel application of *finitization reductions*. A finitization reduction reduces reasoning about an arbitrary-size set to equivalent reasoning about a finite data structure. VizAssert uses finitization reductions to formalize line height computation, margin collapsing, and floating layout.

4.1 Line Heights

In English text, letters are vertically aligned to a *baseline*. On the web, a line of text may mix text of different fonts and sizes, and also images and special inline-block boxes. Web designers may also control the size of the gap between successive lines of text with the line-height CSS property.[3]

The CSS standard requires the browser rendering algorithm to minimize the height of the line while aligning all text and images in that line to a common baseline and while leaving blank space, called *leading*, above and below the text (Figure 4).[4] The foregoing description of line height is not amenable to mechanical reasoning. First, it introduces nested quantifiers by reasoning about the space of possible

[1] Twitter, Facebook, YouTube, Vimeo, Flickr, LinkedIn, Pinterest, Google+, and RSS buttons were present in our evaluation set.

[2] Visual logic expands **if** statements and other standard shorthands to the primitive operators of Figure 3; see Appendix A.

[3] Despite its name, the line-height property does not directly set the height of the line box; instead, it sets the leading. When a line contains images, inline-block boxes, or inline boxes with a different line-height, it can have a different line height than its line-height.

[4] Appendix C.1 reproduces the standard's specification of line heights.

line heights. Second, it describes the set of all objects in a line, a set too complex for efficient mechanical reasoning. VizAssert uses a more efficient description of line heights.

Since each object in the line must be aligned to the baseline, the height of the line must be the highest an object rises above the baseline plus the deepest an object falls below it (plus leading). VizAssert computes the highest an object rises above the baseline with a running maximum (and similarly for the deepest an object falls below the baseline):

$$\mathrm{above\_baseline}(b) :=$$

$$\max \left[\begin{array}{ll} \mathrm{ascent}(b) + \frac{1}{2}\mathrm{leading}(b) & \\ \mathrm{above\_baseline}(b.\mathrm{prev}) & \textbf{if } b.\mathrm{prev} \neq \mathrm{null} \\ \mathrm{above\_baseline}(b.\mathrm{last}) & \textbf{if } b.\mathrm{last} \neq \mathrm{null} \end{array} \right.$$

The first argument to max is the size of the current box (above the baseline); the others pass the above_baseline value left to right along the line and up from children to parents, thus incorporating every box in the line. VizAssert thus reduces line height reasoning to just two values: above_baseline and the symmetric below_baseline.

4.2 Margin Collapsing

In CSS, every box has a *margin*: an area outside the box that other boxes may not intrude into.[5] For most boxes, vertical margins are allowed to overlap if adjacent; Figure 5 demonstrates two margin-collapsing situations. Note that a child's margin may collapse with its parent's, and that the top and bottom margins of zero-height boxes collapse. Due especially to these complex forms of margin collapsing, the set of margins that collapse together may be quite large.

The CSS 2.1 standard describes margin collapsing by describing the conditions under which a pair of margins collapse. Any "connected component" of margins collapses together, and the total margin size is found by adding the most positive and most negative margins in that group. Reasoning about a large set of margins, let alone a set defined by a graph reachability criterion, would be beyond the capabilities of a modern SMT solver. VizAssert thus uses a finitization reduction that replaces the set of collapsed margins by a finite description. VizAssert tracks, instead of the set of collapsed margins, a running maximum of margins seen so far and additional information about those margins. However, doing so for collapsing margins is more complex than the analogous finitization reduction for line height, because it must confront three key problems.

Positive and negative margins The browser rendering algorithm considers positive and negative margins separately, so VizAssert splits the top margin mt into a positive component mt_+ and a negative component mt_-. To track margins as they collapse together, mt_+ is, for any box, the maximum of the top margin mt (if positive), any positive margins of its children that collapse with it, and any positive margins of its previous siblings that collapse with it. Similar accumulations are used for positive and negative top and bottom margins.

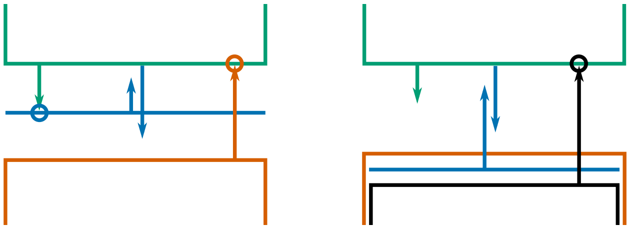


Figure 5. Adjacent vertical margins collapse, and this figure visualizes two possibilities. Each possibility shows several boxes and their margins. Zero-height boxes are represented by horizontal lines. Each arrow represents a margin (all positive) and has the same color as the box that generates it. When a margin determines the location of a box, a circle is drawn around it. The left figure shows how zero-height boxes affect margin collapsing; note that the blue bottom margin is not measured from the bottom of that box because that margin collapses with the box's top margin. The right figure shows a child box's margin (in black) collapsing with its parent's (orange) margin despite being the second sibling of the parent (the first sibling, in blue, is zero height; space has been added around it for clarity). The child box's margin then collapses with a sibling (green) of that parent.

Zero-height first children When a parent's top margin collapses with its first child's, and the first child has zero height, its margins collapse with its next sibling's top margin. This requires propagating information from that next sibling to that first child. This requires additional $\mathrm{mt}^{\uparrow}_{+}$ and $\mathrm{mt}^{\uparrow}_{-}$ values: $\mathrm{mt}^{\uparrow}_{+}$ is equal to mt_+ for boxes with non-zero height, and is equal to the next sibling's $\mathrm{mt}^{\uparrow}_{+}$ for boxes with zero height.

Clearance Some boxes whose clear property is set and which have a nearby floating box have *clearance*. A quirk of the margin collapsing rules declares that if a box of no height has clearance then its margins, and any it collapses with, do not collapse with its parent's bottom margin. An additional boolean field $\mathrm{mb}^{?}$ on each box describes whether its mb_+ and mb_- include margins from such boxes, which is used by the parent to avoid collapsing in those cases.

All told, the finitization reduction for margin collapsing reduces the set of collapsing margins to six real values (mt_+, mt_-, mb_+, mb_-, $\mathrm{mt}^{\uparrow}_{+}$, $\mathrm{mt}^{\uparrow}_{-}$) and the boolean $\mathrm{mb}^{?}$.

4.3 Floating Layout

A floating box is "shifted to the left or right until it touches the edge of its containing box or another floated element" [33]. Floating layout is commonly used to implement sidebars,

[5] This description elides many details; in some cases, other boxes may intrude into a box margin. Appendix C.2 reproduces the standard's specification of margin collapsing.

figures, and menus. Floating layout is particularly challenging and VizAssert's formalization is, as a result, particularly novel. To formalize floating layout, VizAssert reduces the set of all preceding floating boxes to a data structure called an *exclusion zone*. These exclusion zones have minimal *canonical forms* whose compact size makes them an attractive choice for representing floating layout. VizAssert bounds the size of the canonical forms, allowing an SMT solver to use them to compute the positions of floating boxes. Bounding the size of canonical forms is sound: whenever VizAssert adds to a canonical form, it tests whether the bound is hit; if so, VizAssert reruns the verification with a larger bound. Bounding the size of canonical forms is also complete, since no canonical form can be larger than the number of floating boxes on a page.

4.3.1 Floating Layout Semantics

In simple cases, a floating box moves to the left or right edge of its containing box and text wraps around it. However, the rules for positioning multiple floating boxes are quite complex. Floating layout is used for any box whose float property is set, unless its position property is set to absolute or fixed, in which case the box instead uses positioned layout. Appendix C.3 reproduces the standard's specification of floating layout.

The CSS standard describes the positions of floating boxes with nine rules: seven properties that the position must satisfy and two optimization criteria that select among multiple options. For automated reasoning, these rules must be translated to logical formulas that an SMT solver can reason about efficiently. The challenge is two-fold: first, the rules describe relationships that must hold for all pairs of floating boxes, which leads to too many constraints to efficiently solve; and second, encoding the optimization criteria introduces quantifiers, which moves the constraints beyond the capabilities of current SMT solvers. Because of these problems, previous work [39] only allowed restricted uses of floating layout. In contrast, VizAssert supports the full semantics of floating layout using a novel formalization of floating layout based on the exclusion zone data structure.

4.3.2 Exclusion Zones

Laying out a floating box requires knowledge about previous floating boxes. An *exclusion zone* summarizes this information:

- Top and bottom edges of previous floating boxes;
- Right edges of previous left-floating boxes; and
- Left edges of previous right-floating boxes.

This summarization forms a triple (Y, L, R), where Y is the set of y positions of margin top edges of previous floating boxes, L is the set of the coordinates of bottom-right margin corners of previous left floats, and R is the set of the coordinates of bottom-left margin corners of previous right floats.

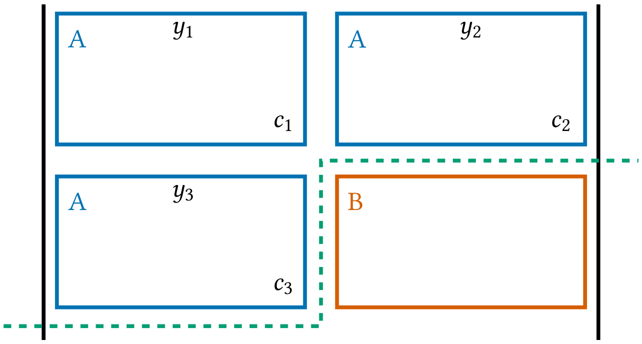


Figure 6. A left-floating boxes is laid out in the topmost, leftmost position outside an *exclusion zone*. Here, the blue boxes labeled "A" are left-floating children of their parent, whose left and right edges are shown. The resulting exclusion zone is all points above (and left of) the green dotted line, which is used to determine the placement of the orange box labeled "B". Formally, the exclusion zone would be a triple $(\{y_1, y_2, y_3\}, \{c_1, c_2, c_3\}, \emptyset)$ or, equivalently, $(\{y_3\}, \{c_3\}, \emptyset)$.

Together, this information describes a region of the screen where a floating box cannot be placed: a box p is excluded from (Y, L, R) when, for some $y \in Y$, $\ell \in L$, and $r \in R$,

$$p.\mathrm{y} < y \lor (p.\mathrm{x} < \ell.\mathrm{x} \land p.\mathrm{y} < \ell.\mathrm{y}) \lor (p.\mathrm{x} > r.\mathrm{x} \land p.\mathrm{y} < r.\mathrm{y})$$

Figure 6 shows the exclusion zone used to lay out a floating box on a web page.

Given the exclusion zone, it is possible to compute the position of a floating box using the box's width w, its containing block p, and the position y^* the box would have if it were laid out with block flow layout. Given this information, the box is then placed such that its margin top edge is at or below y^*, its top corners are not in the exclusion zone, and its top corners are within the containing block (with exceptions for boxes wider than their containing block or of negative width). This leads to an efficient SMT encoding. For example, the following formula constrains y to ensure that both top corners are not in the exclusion zone when the box has positive width but is not wider than its containing block.

$$\begin{aligned}
&0 \le w \le p.\mathrm{width}[\mathrm{content}] \implies w \le r - \ell \\
&r = \min\{x_r \mid (x_r, y_r) \in R \land y < y_r\} \\
&\ell = \max\left(\{x_l \mid (x_l, y_l) \in L \land y < y_l\} \cup \{p.\mathrm{left}[\mathrm{content}]\}\right)
\end{aligned}$$

Similar formulas handle the other two cases and the x position.

The formula above is a property the y position must have. To determine the concrete y position, VizAssert uses the fact that the y position must be a member of $Y \cup \{y^*\}$ and simply tries them in increasing order, to satisfy CSS 2.1's rules that are optimization criteria. Similarly, the x position must (for a left float) be a member of $\{x_l \mid (x_l, y_l) \in L\} \cup \{p.\mathrm{left}[\mathrm{content}]\}$.

4.3.3 Canonical Forms

Two exclusion zones may be *equivalent*—they may exclude the same set of points. For example,

$$e = (\{0\}, \{(0,1), (1,1)\}, \{\}) \text{ and } e' = (\{0\}, \{(1,1)\}, \{\})$$

differ in their L component but exclude the same set of points:

$$(x, y) \in e \Leftrightarrow (x, y) \in e' \Leftrightarrow y < 0 \lor x < 1 \land y < 1$$

A compact canonical form can be computed:

Theorem 4.1. *Every exclusion zone* (Y, L, R) *has a* canonical form (Y^*, L^*, R^*)*, such that:*

- (Y^*, L^*, R^*) *is equivalent to* (Y, L, R)*;*
- *equivalent exclusion zones have identical canonical forms;*
- *among exclusion zones* (Y', L', R') *equivalent to* (Y, L, R)*, the canonical form minimizes* $|Y'| + |L'| + |R'|$*.*

Proof. We construct the canonical form explicitly. Y^* is empty if Y is empty and otherwise is the singleton set

$$Y^* = \{\max(Y \cup Y')\}$$
$$Y' = \{\min\{y_l, y_r\} \mid (x_l, y_l) \in L \land (x_r, y_r) \in R \land x_l > x_r\}$$

L^* is the maximal subset of L such that

$$(x, y) \in L^* \implies \bigwedge_{(x', y') \in L} (x, y) = (x', y') \lor x > x' \lor y > y'$$

R^* is analogous. Checking the three properties given for canonical forms is mechanical. □

Example 4.1. *Floating boxes are frequently used to lay out vertically-aligned items in toolbars and menus. Suppose that the page width is* $w > \ell$*, and consider a layout containing* ℓ *left-floating boxes, all of size* 1×1*. The exclusion zone is* $(\{0\}, \{(1,1), (2,1), \ldots (\ell, 1)\}, \emptyset)$*. This exclusion zone is equivalent to the canonical exclusion zone* $(\{0\}, \{(\ell, 1)\}, \emptyset)$*.*

Because exclusion zones have compact canonical forms, VizAssert can set a bound k on their size; $|L|, |R| \leq 5$ suffices for most web pages. Bounding exclusion zones to k elements raises the challenge of picking a sufficiently large k. VizAssert circumvents this problem by asserting, on every addition of a box to an exclusion zone, that the chosen bound suffices to represent the combined exclusion zone. If this assertion fails for any addition to an exclusion zone, the query is retried with more registers. Since it is sufficient to use as many registers are there are floating boxes on the page, this process will always terminate.

4.3.4 Implementation Using Triples

VizAssert requires an efficient SMT encoding of exclusion zones, including efficient operations for finding the correct position of a floating box and adding a floating box to an exclusion zone.

To compute the position of a float given an exclusion zone (Y, L, R), it is important to first find the float's y position, which is the minimal y position where the float will fit. This requires iterating through L and R togeher and in order of increasing y. To avoid encoding a sort and merge operation in an SMT formula, L and R are stored together. To do so, L and R are stored as a size-k set of triples (y, ℓ, r), where each triple represents $(\ell, y) \in L$ and $(r, y) \in R$. These triples are stored in order of increasing y value, and ℓ (or r) may be missing if L contains no points with y-position greater than y. This encoding allows easily iterating through L and R in order of increasing y. Further, at (y, ℓ, r), the available horizontal space for a floating box is simply $r - \ell$, making it easy to choose the correct vertical position for a new float.

Besides the triples (y, ℓ, r), the set Y must also be encoded. Since VizAssert uses canonical exclusion zones, Y is a singleton set that can be stored directly. When a new float is added to an exclusion zone, Y is updated to the maximum of the current element of Y and the new float's top edge. Values in L and R whose y position is less than the new float's top edge must also be removed. This is easy since L and R are stored in sorted order as triples (y, ℓ, r).

The triple-based encoding significantly shrinks the size of the SMT formulas that define exclusion zone operations and avoids encoding inefficient operations such as sorting and merging. As a further tweak to this encoding, VizAssert allows "redundant triples", which are two triples whose y values differ and whose (ℓ, r) are identical. These redundant triples reduce the size of the SMT formula for adding a float to an exclusion zone, at the cost of possibly requiring more registers. We have found that on balance, redundant triples result in smaller formulas.

4.3.5 Validating the Implementation

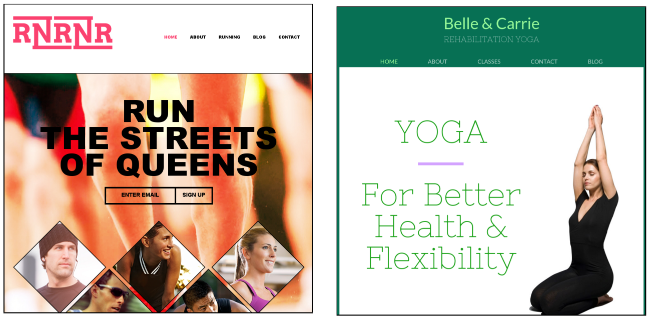
The encoding of exclusion zones is complex. Each time VizAssert runs, it proves that its implementation satisfies the standard's rules for floating layout, for the current k. If the proof fails, VizAssert retains soundness by terminating.

For each of the nine rules required of floating layout, the proof considers the placement of an arbitrary floating box given an arbitrary exclusion zone. Each rule is proven by induction over the exclusion zone's construction: it is proven for an empty exclusion zone, and proven preserved when a floating box is added to an exclusion zone. Each case of the proof is carried out automatically by the SMT solver, since VizAssert's implementation of floating layout is a collection of SMT formulas.

We performed a paper proof that the automated proof always succeeds.

5 Evaluation

VizAssert can verify that realistic web pages satisfy accessibility and usability guidelines. For 62 web pages, VizAssert determined whether it satisfies the 8 general-purpose assertions described in Section 3. We also ran VizAssert on the 6 page-specific assertions and their corresponding web pages.



…organiz[ing] team runs or [...] for any team organizing sports …

…a great fit for any personal health orientated small business …

Figure 7. Two web pages, running and rehabilitation-yoga, from the FWT suite, and the FWT organizers' suggested uses for the templates.

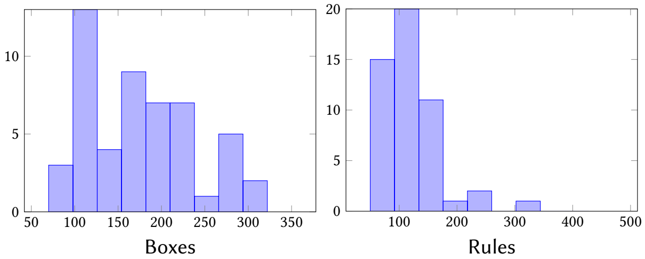


Figure 8. Histogram of the number of boxes and CSS rules for the 62 web pages.

5.1 Subject Web Pages

We collected 62 web pages from an online community of web design professionals that nominates, selects, and publishes high-quality web pages: FreeWebsiteTemplates (FWT) [20]. The published web pages are written by different developers, so they cover a cross-section of common web design techniques. See Figure 7 for examples. Novice web developers use these templates by simply changing the filler text to their own content, while more knowledgeable developers modify the published template or extract and reuse components on their own web pages. Not every web page is expected to pass every assertion, but failures represent departures from web design best practices.

We selected web pages by downloading the 100 most recently published FWT web pages, and retaining the the 62 web pages that fit within the subset of CSS supported by VizAssert (see Section 6). When templates contained multiple pages (for example, a main page, an about page, and a product page) we used only the main page. In aggregate, the 62 web pages average 78 elements (56–93 Inter-Quartile Range), 176 boxes (119–222 IQR), and 128 rules (89–145 IQR). See Figure 8 for histograms of these statistics.

Table 2. Results of verifying accessibility assertions for 62 web pages. T+ and F+ are true and false positives. Overall false positive rate was low: 2.6% of all tests and 17% of counterexamples.

#	Assertion	Verified	T+	F+	Timeout
1	text_size	38	18	3	3
2	interactive_onscreen	59	1	1	1
3	line_width	39	18	3	2
4	accessible_offscreen	[6]36			
5	no_hscroll	60		1	1
6	heading_size	39	21		2
7	overlapping_text	53	2	5	2
8	line_spacing	59	3		
9	contrast	1			
10	no_text_on_bg_image	1			
11	dropdown_offscreen	1			
12	columns_aligned	1			
13	visible_text		1		
14	button_large	1			
Total		388	65	13	11

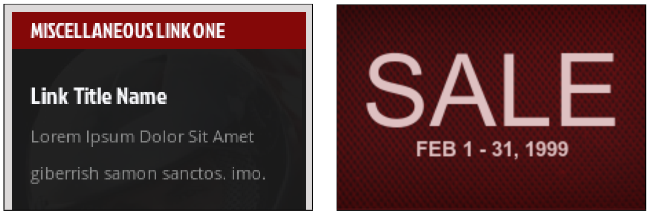
5.2 Results

We ran VizAssert with a timeout of 30 minutes, using a machine with an i7-4790K CPU, 32GB of memory, and Z3 version 4.5.1. VizAssert verified each assertion for all possible screen widths 1024–1920 pixels, screen heights 800–1080 pixels, and default font sizes 16–32 pixels. In total, there were 414 successful verifications, 64 true positives, 13 false positives, and 11 timeouts (see Table 2).

VizAssert outputs rendering parameters that illustrate a violation of design guidelines, so most reports are straightforward to diagnose and fix. For example, the heading_size assertion is violated on the page carracing for a 1856 × 800 browser with 16px text (see Figure 9). The assertion requires more important headings to be larger, but in the page's sidebar, section titles (in second-level h2 headings) are rendered in 14px type, while sidebar links (in third-level h3 headings) are rendered in a larger 16px font. A single-line change to the CSS file fixes this problem by increasing the size of h2 headings.

As an example of a false positive, VizAssert reports that the overlapping_text assertion is violated on the sportinggoods web page for a 1872 × 800 browser with 16px text. The text does not overlap (see Figure 9), but the text boxes do because they reserve space for possible descenders. If the text were "Liquidation" instead of "SALE" then the text would overlap. VizAssert thus detected a failure of the formal assertion, but not a failure of the accessibility guideline that the assertion

[6] Verified on all 62 pages, but 13 verifications are vacuous because no screen-reader elements were identified.


MISCELLANEOUS LINK ONE

Link Title Name

Lorem Ipsum Dolor Sit Amet

giberrish samon sanctos. imo.

SALE

FEB 1 - 31, 1999

Figure 9. On the left, a true positive discovered by VizAssert: the section title, "Miscellaneous Link One", is set in smaller type than subheadings. On the right, a false positive: VizAssert believes that "SALE" can overlap with the date, because it does not reason about the lack of descenders in the letters of "SALE".

formalized. VizAssert would need to reason about the shape of individual letters to avoid this false positive.

VizAssert verifies assertions by transforming them to decidable SMT instances in the theory of quantifier-free linear real arithmetic. The SMT instances are quite large, as shown in Figure 10, due to the complexity of the browser rendering algorithm. Instance size is largely independent of the assertion being verified, instead dependent mostly on the size of the page being verified, since the assertions are small relative to VizAssert's formalization of browser rendering. Proofs of different assertions differ in size, since different assertions may require more or less global reasoning.

Despite the large instances, VizAssert decided most assertions quickly: only 11 executions out of 502 (2.2%) timed out in 30 minutes. Figure 10 plots the time to decide each general-purpose assertion. The line_width assertion takes significantly longer than the others. We believe that this is due to the fact that it requires reasoning about the relationship between three different boxes: a line of text, its first child, and its last child.

5.3 W3C Conformance Tests

VizAssert formalizes novel subsystems of CSS, including the three discussed in Section 4: line height, margin collapsing, and floating layout. An incorrect semantics would lead to VizAssert producing false negatives. Luckily, the W3C provides extensive suites of conformance tests for browser developers to ensure that their browser correctly implements CSS. These tests are provided as web pages containing English-language instructions; to show that VizAssert passes these tests, we follow Cassius [39] in evaluating VizAssert's semantics against these tests by comparing VizAssert's rendering to Mozilla Firefox's.[7]

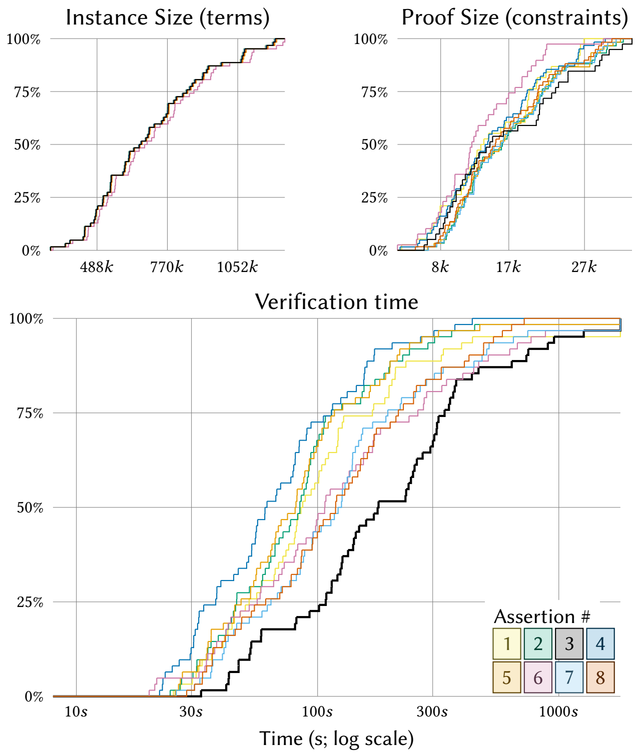


Figure 10. CDF of SMT instance and unsat core for each assertion, and the time required to check satisfiability for the SMT instances.

Table 3. Conformance tests for the seven sections of the CSS 2.1 standard relevant to line height, margin collapsing, and floating layout, all formalized in VizAssert using finitization reductions. Neither tool failed any tests, but Cassius left the overwhelming majority unsupported because its lack of finitization reductions made support for these features impossible.

Section	Title	VizAssert	Cassius	Tests
8.3.0	Margins	318	218	343
8.3.1	Collapsing margins	128	13	133
9.5.0	Floats	184	0	205
9.5.1	Positioning floats	24	0	27
9.5.2	Clearance	60	0	69
10.8.0	Line height	7	1	10
10.8.1	Leading	194	39	219
	Total	915	271	1006

To ensure the correctness of VizAssert's implementation of line height, margin collapsing, and floating layout, we used all W3C conformance tests for subsections 10.8 and 10.8.1 (line height), 8.3 and 8.3.1 (margin collapsing), and 9.5, 9.5.1, and 9.5.2 (floating layout). Table 3 shows results for VizAssert and Cassius. VizAssert passes all but 91 tests;

[7] Minute rounding errors sometimes cause Firefox's rendering of the conformance tests to differ from that mandated by the standard by a fraction of a pixel [39]. We compensate for this in this evaluation by comparing VizAssert's semantics to Firefox's rendering up to an accuracy of a sixth of a pixel.

all 91 of these failing tests use unsupported features, most prominently vertical alignment, right-to-left text, and SVGs, and thus are out of scope for VizAssert's semantics. Among the passing tests are 5 for which VizAssert's rendering is different from Mozilla Firefox's. To verify VizAssert's correctness in these four cases, we manually confirmed that VizAssert's rendering matches the reference rendering; Firefox's rendering of these cases is known to be incorrect [59].

To determine whether finitization reductions were important to achieving an accurate semantics, we compared VizAssert to a semantics without finitization reductions: Cassius [39]. Due to its lack of finitization reductions, most of the conformance tests could not be correctly handled by Cassius.

6 Limitations of VizAssert

CSS is large: dozens of standards totaling thousands of printed pages, VizAssert supports features to enable verification of a substantial fraction of web pages, but it omits many lesser-used features. Some features, such as vertical alignment and right-to-left text, would only require engineering time to implement. Other features, such as formalizing tabular layouts, present significant research challenges in understanding browser implementations, developing finitization reductions, and handling various edge cases.

VizAssert currently handles only HTML and CSS. It would be an interesting challenge to analyze the JavaScript code that modifies web pages (testing accessibility over all possible modifications to the page) or the server-side code that generates pages (testing accessibility over all possible page contents). Temporal operators could be added to visual logic to express accessibility properties for dynamic code [21]. VizAssert could be combined with other tools to achieve this. For example, to verify JavaScriptâĂŹs effect on a web page, a JavaScript static analysis tool could be combined with VizAssert, using the JavaScript analysis to compute the set of possible web page DOMs produced and then using VizAssert to verify their layout properties.

VizAssert addresses only the browser rendering algorithm for CSS. Similar techniques could be applied to other layout algorithms, and finitization reductions similar to those used in VizAssert could be used to formalize those algorithms. For example, exclusion zones may be useful for formalizing the behavior of floats in $\TeX$, and the line height reduction could apply to text layout in iOS or Android applications.

7 Related Work

VizAssert builds on previous efforts to formally study visualizations and graphical interfaces. It adds to the suite of tools available to web developers to validate and improve their web pages.

7.1 Formalization of Browser Rendering

VizAssert's formalization of the browser rendering algorithm is significantly more detailed and conformant than that of previous work, Cassius [39], on which it builds. (Of the 13 major subsystems of the VizAssert formalization in Figure 2, seven are not present in Cassius.) VizAssert's advances on this prior work include not only the finitization reductions used to formalize line height, margin collapsing, and floating layout, but also several additional CSS properties not supported by Cassius and richer CSS selector support.

None of the realistic web pages used to evaluate VizAssert fit within the small subset of CSS implemented by Cassius. Besides the subsystems discussed in Section 4, these pages also make frequent use of positioned layout (which allows placing a box at a fixed pixel position on the screen) and clearance (which allows moving a box vertically in response to floats). Support for the text-indent and list-style-position properties is also added in VizAssert.

Cassius was intended to be used for synthesizing CSS from examples, and thus cannot assume a known, concrete CSS file. When verifying web pages, however, the CSS file is known; this allows VizAssert to compute selector matching and cascading outside the SMT solver, making it possible to support descendant, child, and pseudo-class selectors that Cassius, which did selecting and matching in the SMT solver, could not support. The assumption of a known CSS file also allows VizAssert to soundly handle the em and ex units, which scale with the font size and are thus essential for writing accessible web pages.

Other work has formalized subsets of CSS using attribute grammars [32]. Due to limitations of attribute grammars, these efforts cover a smaller subset of CSS than VizAssert, though a bigger one than Cassius. Furthermore, these formalizations are not meant for verification, only rendering, and thus can not be used for mechanical reasoning.

7.2 Formalized Visual Reasoning

Previous work has approached the idea of mechanical reasoning about visual layout from many directions. Some authors have investigated synthesizing programs from visual manipulations [12, 23]. Others have developed domain-specific languages for visual layout, such as the grammar of graphics [62] used by ggplot2 [61]. ConstraintSS [4] and similar work on constraint-based layout [6, 22, 47, 51, 64] allow specifying layouts via a set of constraints, synthesizing the layout from the constraints. While all of this work involves some formal reasoning about layout, none of it provides a way to assert or verify accessibility or usability properties.

Other work has explored testing disciplines for graphical interfaces. Sikuli Test [10] provides a simple scripting language for describing sequences of user inputs and for comparing the interface pixel-for-pixel to a fixed image. Other authors [26, 30, 37] provide techniques for capturing such scripts from user interactions. This work has demonstrated the importance of developing the right formal descriptions of correct graphical interface behavior [63]. VizAssert incorporates this insight by developing an expressive logic in which developers can specify their accessibility and usability properties.

One such expressive logic is provided by Cornipickle [21], which tests, for particular rendering parameters and sequence of user actions, temporal properties of interactive web pages. Visual logic is inspired by Cornipickle, but is adapted to enable automated reasoning: visual logic adds functions for traversing the tree of boxes, adds the ancestor function for relating distant boxes, and limits multiplication to linear arithmetic. On the other hand, since VizAssert does not reason about JavaScript, it omits Cornipickle's temporal operators. Cornipickle does not offer a verification tool like VizAssert for its assertion language; it simply offers a convenient method for testing assertions on particular renderings.

7.3 Accessibility

Major GUI frameworks, including Android, iOS, OS X, and the multiple Windows frameworks, include support for accessibility information, which often involves annotating widgets with roles or describing their content. For the web, this framework is ARIA [55], which defines the meaning of accessibility attributes, and the WCAG [53], which documents best practices for their use. Tools such as WAVE [60], SOAtest [40], and 508checker [18] check that the web page HTML satisfies accessibility best practices such as avoiding justified text and having meaningful captions. Academic work has also been done on this problem. Raven [17] applies "validation rules" to ensure that the Java objects that make up a graphical interface properly contain captions, summaries, and valid mnemonics, important tests for accessible interfaces. Chang, Yeh, and Miller extend pixel-based tools such as Sikuli to access accessibility information, such as content and role information for user interface elements [9]. All of these tools only test visual layout for particular rendering parameters, so do not provide strong guarantees independent of the renderings parameters.

7.4 Tools for Interface Developers

Many recent papers have developed tools for designers of user interfaces. Bricolage [25] uses heuristics to transfer styles from one page to another; SeeSS [27] tracks the effects of CSS changes; ReVision [45] extracts data from visualizations and synthesizes new presentations for that data; and Remaui [36] uses heuristics to synthesize Android layouts from mockups. These tools aim to simplify the task of writing or editing graphical interfaces; VizAssert focuses on verifying these interfaces' accessibility and usability. Some authors have developed heuristics for detecting problematic layouts [5, 28, 57, 58]. Other tools attempt to detect cross-browser differences for web pages, often using heuristics to build a high-level model of the page behavior [11, 31, 44] These tools are useful, but are each specific to a single web page property; VizAssert allows web developers to write and verify custom assertions.

8 Conclusion

Verifying visual properties of user interfaces is important to ensure applications are usable and accessible to all users. We developed a visual logic of assertions and a publicly-available automated tool, VizAssert, that verifies visual assertions for web pages regardless of user preferences and screen sizes. Building VizAssert contains novel formalizations of multiple aspects of the CSS layout algorithm, including line height, margin collapsing, and floating layout, implemented using novel finitization reductions. For a set of real-world webpages, VizAssert verified assertions from usability guidelines and standards, finding 64 errors with only 13 false positives.

Acknowledgments

We thank Jen Mankoff and Anat Caspi for invaluable comments on early drafts. We also thank our shepherd Ben Livshits and the anonymous reviewers for their valuable suggestions.

This work was supported by an Adobe Fellowship and by gifts from Adobe. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1256082. This material is based upon work supported by the United States Air Force under Contract No. FA8750-15-C-0010, and on research sponsored by Air Force Research Laboratory and DARPA under agreement number FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Apple. 2017. Human Interface Guidelines. https://developer.apple.com/ios/human-interface-guidelines/

[2] Apple. 2017. UI Design Do's and Don'ts. https://developer.apple.com/design/tips/

[3] Apple Developer. 2017. UIKit Framework. https://developer.apple.com/documentation/uikit

[4] Greg J. Badros, Alan Borning, Kim Marriott, and Peter J. Stuckey. 1999. Constraint Cascading Style Sheets for the Web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST'15)*. ACM, 73–82. https://doi.org/10.1145/320719.322588

[5] Jeffrey P. Bigham. 2014. Making the Web Easier to See with Opportunistic Accessibility Improvement. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, 117–122. https://doi.org/10.1145/2642918.2647357

[6] Alan Borning, Richard Lin, and Kim Marriott. 1997. Constraints for the Web. In *Proceedings of the Fifth ACM International Conference on Multimedia (MULTIMEDIA '97)*. ACM, 173–182. https://doi.org/10.1145/266180.266361

[7] Matthew Butterick. 2010. *Practical Typography*. Matthew Butterick Typography, online only.

[8] Lyndon Cerejo. 2011. A User-Centered Approach to Web Design for Mobile Devices. https://www.smashingmagazine.com/2011/05/a-user-centered-approach-to-web-design-for-mobile-devices

[9] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 2011. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, 245–256. https://doi.org/10.1145/2047196.2047228

[10] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, 1535–1544. https://doi.org/10.1145/1753326.1753555

[11] S. R. Choudhary, M. R. Prasad, and A. Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 171–180. https://doi.org/10.1109/ICST.2012.97

[12] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 341–354. https://doi.org/10.1145/2908080.2908103

[13] Burns David. 2012. *Selenium 2 Testing Tools: Beginner's Guide*. Packt Publishing, Birmingham, UK.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[15] Diet. 2000. Basic Act on the Formation of an Advanced Information and Telecommunications Network Society. http://japan.kantei.go.jp/it/it_basiclaw/it_basiclaw.html

[16] European Commission. 2016. Directive (EU) 2016/2102 of the European Parliament and of the Council of 26 October 2016 on the accessibility of the websites and mobile applications of public sector bodies. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2016.327.01.0001.01.ENG&toc=OJ:L:2016:327:TOC

[17] Barry Feigenbaum and Michael Squillace. 2006. Accessibility Validation with RAVEN. In *Proceedings of the 2006 International Workshop on Software Quality (WoSQ '06)*. ACM, 27–32. https://doi.org/10.1145/1137702.1137709

[18] LLC Formstack. 2014. Free Section 508 Compliance Checker. http://www.508checker.com/check

[19] Amy Fowler. 2017. A Swing Architecture Overview. http://www.oracle.com/technetwork/java/architecture-142923.html

[20] Free Website Templates. 2017. Free Website Templates. https://freewebsitetemplates.com

[21] Sylvain Hallé, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. 2015. Testing Web Applications Through Layout Constraints. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, IEEE, 1–8.

[22] Osamu Hashimoto and Brad A. Myers. 1992. Graphical Styles for Building Interfaces by Demonstration. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology (UIST '92)*. ACM, 117–124. https://doi.org/10.1145/142621.142635

[23] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST'14)*. ACM, 231–241. https://doi.org/10.1145/2642918.2647378

[24] Melody Ivory and Aline Chevalier. 2002. *A Study of Automated Web Site Evaluation Tools*. Technical Report. University of Washington, Department of Computer Science.

[25] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-based Retargeting for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'11)*. ACM, 2197–2206. https://doi.org/10.1145/1978942.1979262

[26] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Automated Generation of Visual Web Tests from DOM-based Web Tests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. ACM, 775–782. https://doi.org/10.1145/2695664.2695847

[27] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. 2013. 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 (UIST '13)*. ACM, 353–356. https://doi.org/10.1145/2501988.2502006

[28] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated Repair of Layout Cross Browser Issues Using Search-based Techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, 249–260. https://doi.org/10.1145/3092703.3092726

[29] Jennifer Mankoff, Holly Fait, and Tu Tran. 2005. Is Your Web Page Accessible?: A Comparative Study of Methods for Assessing Web Page Accessibility for the Blind. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*. ACM, 41–50. https://doi.org/10.1145/1054972.1054979

[30] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03)*. IEEE Computer Society, 260–. http://dl.acm.org/citation.cfm?id=950792.951350

[31] A. Mesbah and M. R. Prasad. 2011. Automated cross-browser compatibility testing. In *2011 33rd International Conference on Software Engineering (ICSE)*. 561–570. https://doi.org/10.1145/1985793.1985870

[32] Leo A. Meyerovich and Rastislav Bodik. 2010. Fast and Parallel Webpage Layout. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, 711–720. https://doi.org/10.1145/1772690.1772763

[33] Mozilla Developer Network. 2017. float. https://developer.mozilla.org/en-US/docs/Web/CSS/float

[34] Mozilla Developer Network. 2017. Mobile accessibility checklist. https://developer.mozilla.org/en-US/docs/Web/Accessibility/Mobile_accessibility_checklist

[35] National Federation for the Blind. 2016. Blindness Statistics. https://nfb.org/blindness-statistics

[36] Tuan A. Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with REMAUI. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE'15)*. IEEE, 248–259.

[37] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. 1998. A Visual Test Development Environment for GUI Systems. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)*. ACM, 82–92. https://doi.org/10.1145/271771.271793

[38] Jason Pamental. 2014. A More Modern Scale for Web Typography. http://typecast.com/blog/a-more-modern-scale-for-web-typography

[39] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 181–194. https://doi.org/10.1145/2983990.2984010

[40] Parasoft. 2017. Web UI Testing. https://www.parasoft.com/capability/web-ui-testing/

[41] Pearson. 2017. Making E-Learning Accessible. http://wps.pearsoned.com/accessibility/

[42] Android Open Source Project. 2017. UI Overview. https://developer.android.com/guide/topics/ui/overview.html

[43] Murray Rowan, Peter Gregor, David Sloan, and Paul Booth. 2000. Evaluating Web Resources for Disability Access. In *Proceedings of the Fourth International ACM Conference on Assistive Technologies (Assets '00)*. ACM, 80–84. https://doi.org/10.1145/354324.354346

[44] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICSM.2010.5609723

[45] Manolis Savva, Nicholas Kong, Arti Chhajta, Li Fei-Fei, Maneesh Agrawala, and Jeffrey Heer. 2011. ReVision: Automated Classification, Analysis and Redesign of Chart Images. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, 393–402. https://doi.org/10.1145/2047196.2047247

[46] Terry Sullivan and Rebecca Matson. 2000. Barriers to Use: Usability and Content Accessibility on the Web's Most Popular Sites. In *Proceedings on the 2000 Conference on Universal Usability (CUU '00)*. ACM, 139–144. https://doi.org/10.1145/355460.355549

[47] Ivan E. Sutherland. 1964. Sketch Pad a Man-machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop (DAC '64)*. ACM, 6.329–6.346. https://doi.org/10.1145/800265.810742

[48] Anthony T. 2012. Finger-friendly design: ideal mobile touchscreen target sizes. https://www.smashingmagazine.com/2012/02/finger-friendly-design-ideal-mobile-touchscreen-target-sizes/

[49] US DOJ. 2010. Department of Justice Advanced Notice of Proposed Rulemaking, RIN 1190-AA61. https://www.ada.gov/anprm2010/web%20anprm_2010.htm

[50] US DOJ. 2017. ADA Best Practices Tool Kit for State and Local Governments. https://www.ada.gov/pcatoolkit/chap5toolkit.htm

[51] Christopher J. van Wyk. 1982. A High-Level Language for Specifying Pictures. *ACM Trans. Graph.* 1, 2 (April 1982), 163–182. https://doi.org/10.1145/357299.357303

[52] Minh N. Vu and Susan Ryan. [n. d.]. 2017 Website Accessibility Lawsuit Recap: A Tough Year for Businesses. https://www.adatitleiii.com/2018/01/2017-website-accessibility-lawsuit-recap-a-tough-year-for-businesses/

[53] W3C. 2008. Web Content Accessibility Guidelines 2.0. https://www.w3.org/TR/WCAG/

[54] W3C. 2011. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. https://www.w3.org/TR/2011/REC-CSS2-20110607/

[55] W3C. 2016. WAI-ARIA Overview. https://www.w3.org/WAI/intro/aria

[56] W3Schools. 2017. CSS Dropdowns. https://www.w3schools.com/css/css_dropdowns.asp

[57] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2017. Automated Layout Failure Detection for Responsive Web Pages Without an Explicit Oracle. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, 192–202. https://doi.org/10.1145/3092703.3092712

[58] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. 2015. Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 709–714. https://doi.org/10.1109/ASE.2015.31

[59] Web Platform Tests. 2017. web-platform-tests dashboard, WPT/css/CSS2/floats. https://wptdashboard.appspot.com/css/CSS2/floats

[60] WebAIM. 2017. WAVE Web Accessibility Tool. http://wave.webaim.org/

[61] Hadley Wickham. 2009. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, New York, New York, USA. http://ggplot2.org

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

[63] Qing Xie and Atif M. Memon. 2007. Designing and Comparing Automated Test Oracles for GUI-based Software Applications. *ACM Trans. Softw. Eng. Methodol.* 16, 1, Article 4 (Feb. 2007), 38 pages. https://doi.org/10.1145/1189748.1189752

[64] Brad Vander Zanden and Brad A. Myers. 1991. The Lapidary Graphical Interface Design Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*. ACM, 465–466. https://doi.org/10.1145/108844.109005

A Visual Logic Semantics

Visual logic describes properties of a *box trees*, which is collection of colored rectangles arranged in a tree. No assumptions are made about the rectangles or their colors; for example, rectangles are not required to be contained within their parent rectangle.

Given a box tree, a visual logic assertion is either true or false, and can be checked by simply checking whether the conditional defining the assertion is true for all choices of n boxes (for an assertion that quantifies over n boxes). Visual logic terms describe one of four types: $\langle cond \rangle$ terms denote to booleans, $\langle real \rangle$ terms denote to real numbers,[8] $\langle color \rangle$ terms denote to triples of real numbers or the special transparent value, and $\langle box \rangle$ terms denote to nodes in the box tree, which are assumed to have child, sibling, and parent pointers, types (and, for boxes with text type, textual content as a string), positions for their top, right, bottom, and left edges (as reals), and a color (as a triple of reals or as the transparent value). The denotation of most terms follows the standard meaning of that term. Some terms have no sensible meaning, such as "transparent.r" or "null.top". Visual logic does not assign such terms any meaning, treating them as invalid, and VizAssert behaves unpredictably given such terms. This section describes additional details for a few constructs.

The "$\langle box \rangle$.type" construct describes the box's type: the root box is a window; inline boxes define text formatting; line boxes lay out text in lines; text is raw text; and all others are block boxes. These types and their meaning are defined by the CSS browser rendering algorithm; in assertions, the text type is useful for identifying textual content. Actually reading the textual content is challenging since text can be split into multiple lines. The provided "$\langle box \rangle$.whitespace" construct, which is valid only for text boxes, determines whether a text box contains only whitespace; note that this property is not affected by line breaking.

Real numbers can be given directly as constants, added, or multiplied. However, note that multiplication "$\mathbb{R} \times \langle real \rangle$" requires the left hand argument to be a constant, thereby forcing real expressions to be linear. The actual VizAssert implementation does not enforce this requirement, but in most cases using non-constants on both sides of a multiplication causes the underlying solver to report unknown, which VizAssert then displays to the developer. Visual logic assertions that adhere to the restriction to linear arithmetic are guaranteed to be decidable.

Colors can be specified as constants, or as the foreground or background color of a box. Colors other than the transparent value are represented by triples (r, g, b) of real numbers in the range $[0, 1]$. Foregrounds and backgrounds can be γ-corrected, a color-theoretic operation that transforms a color (r, g, b) to $(r^{2.2}, g^{2.2}, b^{2.2})$. This odd operation and unusual exponent relates the RGB triple stored in the computer to the intensity of light emitted from the red, green, and blue diodes on the computer's display.[9] The uncorrected colors are useful because they correspond to values in the CSS file; the corrected colors are useful for calculations of contrast, intensity, or color match. Since SMT solvers cannot reason efficiently about exponentiation, VizAssert precomputes gamma correction for all colors mentioned in the CSS file, which are the only colors that can be the foreground or background of a box. This approach is why visual logic does not allow gamma correcting arbitrary colors constructed with the rgb construct.

To make writing visual logic easier, the following common abbreviations are used:

1. $b.\text{width}[e] \equiv b.\text{right}[e] - b.\text{left}[e]$
2. $b.\text{height}[e] \equiv b.\text{bottom}[e] - b.\text{top}[e]$
3. $x \leq y \equiv x < y \lor x = y$
4. $x \geq y \equiv x > y \lor x = y$
5. $x \neq y \equiv \neg(x = y)$
6. $P(x_{\{a,b\}}) \equiv P(x_a) \land P(x_b)$

We also use **let** and **if** as shorthand for more complex queries:

$$\textbf{let } v = \mathcal{X} \textbf{ in } \mathcal{Y}[v] \equiv \mathcal{Y}[\mathcal{X}]$$

and

$$\mathcal{E}[\textbf{if } C \textbf{ then } \mathcal{A} \textbf{ else } \mathcal{B}] \equiv \\ (\mathcal{E}[C] \implies \mathcal{E}[\mathcal{A}]) \land (\mathcal{E}[\neg C] \implies \mathcal{E}[\mathcal{B}])$$

B Formalizing Guidelines

This section demonstrates the expressiveness of visual logic via 14 examples. Each example starts from a guideline from Table 1 and then adapts the guideline to the evaluated web pages and expresses that guideline as an assertion in visual logic. Of the 14 guidelines, 8 were formalized so that the same assertion was meaningful on all 62 evaluation web pages, while 6 had to be formalized in a way that was specific to a single page. Appendix B.1 describes the general-purpose assertions, while Appendix B.2 covers page-specific assertions.

B.1 General-Purpose Assertions

1. Text is at least 14px tall

Users with difficulty seeing cannot read small text, so page contents should be at least 14 pixels tall. This requirement can be expressed in visual logic:

$$\forall b \in \mathcal{B} : \text{page\_content}(b) \implies b.\text{height} \geq 14$$

Some text is page content, necessary for using the web page, and other text is not. First, text boxes that contain only whitespace are invisible to sighted users, and thus unlikely to be

[8] In fact, rationals suffice if all constants are rational, making the representation computable.

[9] Gamma correction is thus a property of the display, and can vary between devices; however, VizAssert supports only the most common exponent of 2.2.

essential page content.[10] Second, copyright and other legal notices are often mechanically-generated, present on every page, and necessary for legal reasons, not because they are a functional part of the page. These notices were always located in page footers, which could be identified by the "#footer" or ".footer" selectors.

$$\begin{aligned}\mathsf{page\_content}(b) = b.\mathsf{type} = \mathsf{text} \wedge \neg b.\mathsf{whitespace} \\ \wedge \neg(\mathsf{is\_descendant}(b, (\#\mathsf{footer}, .\mathsf{footer})))\end{aligned}$$

These two criteria were sufficient for the 62 evaluated pages.

The is_descendant function determines whether a box is a descendant of an element matching a certain selector:

$$\mathsf{is\_descendant}(b, S) = b.\mathsf{ancestor}(? \in \$(S)) \neq \mathsf{null}$$

2. Text can be resized by up to 200%

Users with difficulty seeing often increase the default font size, but some pages are not functional when this setting is changed. VizAssert checks assertions for font size preferences between 16 pixels, the default, and 32 pixels, capturing the 200% resizing behavior. All that remains to test is that the page remains usable despite the resizing. After examining the 62 evaluation web pages, we determined that the most important function of the web pages was that interactive elements are on-screen and thus accessible:

$$\forall b \in \mathcal{B} : \mathsf{interactive}(b) \implies \mathsf{onscreen}(b)$$

We found that links, input fields, and buttons were the most common interactive elements: $\mathsf{interactive}(b) = b \in \$(\mathsf{a}, \mathsf{input}, \mathsf{button})$. To be off-screen, a box must be either to the left or above the root box; boxes to the right or below the root box can be scrolled to:[11]

$$\mathsf{onscreen}(b) = b.\mathsf{left} \geq \mathsf{root.left} \wedge b.\mathsf{top} \geq \mathsf{root.top}$$

3. Lines are no more than 80 characters

Users with difficulty seeing sometimes find it hard to keep their place in a long line text. The 62 evaluation pages are all in English, where 80 characters is commonly used rule of thumb for line length:

$$\begin{aligned}\forall b \in \mathcal{B} : \mathsf{page\_content}(b) \implies \\ \mathsf{line\_width}(\mathsf{line}(b)) \leq 80 \times \mathsf{char\_width}(b)\end{aligned}$$

The ancestor function is used to select the line containing a text box:

$$\mathsf{line}(b) = b.\mathsf{ancestor}(?.\mathsf{type} = \mathsf{line})$$

To count characters, we use a proportional conversion from character height to character width, $\mathsf{char\_width}(b) = \frac{10}{19} \times b.\mathsf{height}$. The line width refers to the distance between the first and last bit of text in the line:

$$\mathsf{line\_width}(\ell) = \ell.\mathsf{last\text{-}child.right} - \ell.\mathsf{first\text{-}child.left}$$

The line box itself is wider in CSS—it covers the space available for text, even if unused.

4. Elements for screen-reader users are off-screen

Users who are partially or fully blind often navigate the web using a screen-reader. Since screen-reader users cannot see the page, web developers often add content for screen-reader users and position that content off-screen.

$$\forall b \in \mathcal{B} : \mathsf{for\_screenreader}(b) \implies \neg\mathsf{onscreen}(b)$$

38 of the 62 evaluation pages featured social sharing buttons,[12] implemented with background images representing social networks' logos. Since background images are not accessible to blind users, these pages also included text naming the service, but positioned this text off-screen.

$$\mathsf{for\_screenreader}(b) = b.\mathsf{type} = \mathsf{text} \wedge \mathsf{is\_descendant}(b, \#S_i)$$

where $S_1, S_2, \cdots$ are social services used.

5. The page does not require horizontal scrolling

Users with sensorimotor disabilities (and mobile users using one hand) may find scrolling in two dimensions difficult. Most pages should thus stick to vertical scrolling. Browsers enable horizontal scrolling when a box extends past the right-hand edge of the root box:

$$\forall b \in \mathcal{B} : b.\mathsf{right} \leq \mathsf{root.right}$$

This assertion is checked for browsers 1024–1920 pixels wide and 800–1024 pixels tall since the evaluation pages seem designed for laptop displays.

6. Headings must form a visual hierarchy

Web designers use visual details, like size or spacing, to suggest to sighted users which content on the page is more or less important. Users who browse the web with a screen-reader instead read the hierarchical heading tags h1–h6, where smaller numbers describe more important content. However, web page developers can misuse these tags, making screen-reader users and sighted users perceive a different hierarchy of importance in page content. To avoid this, lower-numbered headings should be more visually important than higher-numbered headings:

$$\begin{aligned}\forall b_1, b_2 \in \mathcal{B} : \mathsf{in\_header}(b_{\{1,2\}}) \wedge \\ \mathsf{header\_level}(b_1) < \mathsf{header\_level}(b_2) \implies \\ \mathsf{visual\_importance}(b_1) > \mathsf{visual\_importance}(b_2)\end{aligned}$$

Visual importance can be established by size, spacing, or even position, but on the 62 evaluation web pages, it was

[10] It is fairly common to generate such boxes accidentally by formatting and indenting HTML.

[11] Note that for web pages, y-values increase down the page, with the origin at the top.

[12] Twitter, Facebook, YouTube, Vimeo, Flickr, LinkedIn, Pinterest, Google+, and RSS were in our evaluation set.

most often established by text size: visual_importance(b) = b.height. Selecting headings is done using the is_descendant:

$$\text{in\_header}(b) = \text{page\_content}(b) \wedge \text{is\_descendant}(b, (\text{h1}, \text{h2}, \text{h3}, \text{h4}, \text{h5}, \text{h6}))$$

Once a heading is selected, its heading can be computed by testing its tag name:

$$\text{header\_level}(b) = \textbf{if } b \in \$(\text{h1}) \textbf{ then } 1 \textbf{ else } \ldots$$

7. Text should not overlap

Text that overlaps other text is difficult to read. Thus text that is meant to be read should not overlap other text:

$$\forall b_1, b_2 \in \mathcal{B} : \text{page\_content}(b_{\{1,2\}}) \implies \neg\text{overlaps}(b_1, b_2)$$

Two boxes overlap when they are both horizontally and vertically adjacent.

$$\text{overlaps}(b_1, b_2) = \text{h\_adjacent}(b_1, b_2) \wedge \text{v\_adjacent}(b_1, b_2)$$

Horizontal adjacency tests three separate cases:

$$\text{h\_adjacent}(b_1, b_2) = \bigvee \begin{cases} b_1.\text{right} > b_2.\text{left} > b_1.\text{left} \\ b_2.\text{right} > b_1.\text{left} > b_2.\text{left} \\ b_1.\text{left} = b_2.\text{left} \end{cases}$$

Due to rounding error when measuring the ascent and descent of letters in a font, text boxes may overlap by a fraction of a pixel. Vertical adjacency therefore includes an additional tweak: text boxes must vertically overlap by a pixel or more to be considered overlapping.

8. Lines should be not be densely spaced

Lines that are spaced too closely (even if the text itself does not overlap) make it difficult to keep one's place in line.

$$\forall b_1, b_2 \in \mathcal{B} : \text{page\_content}(b_{\{1,2\}}) \wedge \text{line}(b_1).\text{next} = \text{line}(b_2) \implies b_2.\text{top} - b_1.\text{top} \geq c \times b_1.\text{height}$$

This assertion selects two text boxes from sibling lines and ensures that the vertical gap between them is at least c times the height of the taller text box. We set c to 1.04—smaller than most guidelines recommend—because it seemed that many of the evaluated pages used tight line spacing for visual effect.

B.2 Page-Specific Assertions

In translating the guidelines of Table 1, some assertions had to refer to page-specific elements or were otherwise limited to a single web page in our test suite. Since web developers tend to work on single pages, not evaluation suites of dozens of them, these single-page assertions better model the expected use case of VizAssert.

9. Text has sufficient contrast

Users with difficulty seeing can have difficulty making out text against a similarly-colored background, and color-blind users may see different colors as similar. Pages should give text high contrast against their background in order to make their text more readable for these users.

$$\forall b \in \mathcal{B} : \text{page\_content}(b) \implies \text{good\_contrast}(\gamma(b.\text{fg}), \gamma(b.\text{bg}^*))$$

This computation uses gamma-corrected colors since it is the contrast of the light shown by the monitor that is relevant here.

A ratio between $\frac{1}{3}$ and 3 indicates text with low contrast, a challenge for visually-impaired users, especially those with poor color vision:

$$\text{high\_contrast}(\text{lum}_1, \text{lum}_2) = \bigvee \begin{cases} \text{lum}_1 \geq 3 \times (\text{lum}_2) \\ \text{lum}_2 \geq 3 \times (\text{lum}_1) \end{cases}$$

This luminance is computed (by a standard formula [53]) for the foreground and background colors of the text.

$$\text{good\_contrast}(fg, bg) = fg \neq \text{transparent} \wedge \text{high\_contrast}(\text{lum}(fg) + .05, \text{lum}(bg) + .05)$$

Most boxes have transparent backgrounds, allowing an ancestor's background to show through; the text's background thus comes from its ancestor:

$$b.\text{bg}^* = (a.\text{bg } \textbf{if } a \textbf{ else } \text{white}) \textbf{ where } a = b.\text{ancestor}(?.\text{bg} \neq \text{transparent})$$

Unfortunately, this assertion only fits pages where text is placed above solid color backgrounds. In our evaluation suite, this describes only the rehabilitation-yoga web page. Support for background images is challenging since a background image may have different contrast at different points. As demonstrated by the next example, this challenge can be avoided in certain cases.

10. Text does not overlap image

Not only do visually-disabled users have difficulty seeing low-contrast text, they may also have difficulty seeing text placed over a busy background image. For example, the web page puppy features a background image with a puppy. Text is then placed atop a blank portion of this image.

$$\forall b_1, b_2 \in \mathcal{B} : b_1 \in \$(\#\text{background}) \wedge \text{over\_bg}(b_2) \implies \neg\text{over\_img}(b_1, b_2)$$

The background image can be identified by the background identifier:

$$\text{over\_bg}(b) = b_2.\text{type} = \text{text} \wedge \text{is\_descendant}(b, \#\text{background})$$

The puppy occupies a 300 square pixel area at the right hand edge of the background image:

$$\text{over\_img}(b_1, b_2) = b_1.\text{right}[\text{padding}] - b_2.\text{right} \geq 300$$

11. Dropdown menus are hidden when not selected

Visual logic can also express usability properties not specifically related to the needs of disabled users. For example, one web page in the evaluation set, gardenwalkthrough, used drop-down menus, where the menu was hidden off-screen[13] when not dropped down. Since no dropdown is selected by default, it is enough to assert that all boxes selected by this selector are off-screen.

$$\forall b \in \mathcal{B} : b \in \$(\mathsf{\#header > ul > li > ul}) \implies \neg\mathsf{onscreen}(b)$$

In this assertion, the selector matches dropdown menus.

12. Columns are vertically aligned

Visual logic can also be used the verify purely esthetic assertions. The web page tailorshopwebsitetemplate uses a multi-column layout; columnar layouts are more attractive when the tops of the columns align and when the columns themselves do not overlap:

$$\forall b_1, b_2 \in \mathcal{B} : b_1 \in \$(s_\ell) \wedge b_2 \in \$(s_r) \implies \mathsf{vertically\_aligned}(b_1, b_2) \wedge \neg\mathsf{overlap}(b_1, b_2)$$

In this assertion, s_ℓ selects the left column and s_r selects the right column. On tailorshopwebsitetemplate, columns are vertically aligned at their margin edges:

$$\mathsf{vertically\_aligned}(b_1, b_2) = b_1.\mathsf{top}[\mathsf{margin}] = b_2.\mathsf{top}[\mathsf{margin}]$$

13. Full link text should be visible

On the genericwebsitetemplate, there are several links asking the user to "Click here for more information". However, the text is inside a box that hides any content that overflows its boundaries. It's important that none of the text in the link is cut off, since the text would become difficult to read and eventually would disappear.

$$\forall b \in \mathcal{B} : \mathsf{descends}(b, S) \wedge \mathsf{page\_content}(b) \implies \mathsf{within}(b, b.\mathsf{ancestor}(? \in \$(S)))$$

The selector S = (.body ul li) identifies the boxes that these captions are located in. The within predicate checks each edge of the box:

$$\mathsf{within}(b_1, b_2) = \bigwedge \left[\begin{array}{l} b_1.\mathsf{left} \geq b_2.\mathsf{left} \\ b_1.\mathsf{top} \geq b_2.\mathsf{top} \\ b_1.\mathsf{right} \leq b_2.\mathsf{right} \\ b_1.\mathsf{bottom} \leq b_2.\mathsf{bottom} \end{array} \right.$$

Unfortunately, the assertion is violated at some default font sizes; VizAssert finds this assertion violation and provides a counterexample.

14. Main button should be large

Users with motor disabilities have difficulty using small buttons; users on mobile devices have similar problems. Common recommendations [48] suggest that buttons be at least 24 pixels tall and 72 pixels wide (on mobile devices, the wide button means it will not be entirely occluded by the user's finger). Small, infrequently-used buttons may have be small, but the main button on a page (the "call to action") must be easy to click for all users. On the carrepairshop web page, that call to action is the "Book an Appointment" button:

$$\forall b \in \mathcal{B} : b \in \$(S) \implies b.\mathsf{width} \geq 72 \wedge b.\mathsf{height} \geq 24$$

The selector S = (#body .body div > a) selects the "Book an Appointment" button.

C Standards Text

C.1 Line Height

The CSS 2.1 specification defines the rules for line height in section 10.8 [54]:

> The height of a line box is determined as follows:
> 1. The height of each inline-level box in the line box is calculated. For replaced elements, inline-block elements, and inline-table elements, this is the height of their margin box; for inline boxes, this is their 'line-height'. (See "Calculating heights and margins" and the height of inline boxes in "Leading and half-leading".)
> 2. The inline-level boxes are aligned vertically according to their 'vertical-align' property. In case they are aligned 'top' or 'bottom', they must be aligned so as to minimize the line box height. If such boxes are tall enough, there are multiple solutions and CSS 2.1 does not define the position of the line box's baseline (i.e., the position of the strut, see below).
> 3. The line box height is the distance between the uppermost box top and the lowermost box bottom. (This includes the strut, as explained under 'line-height' below.)
>
> Empty inline elements generate empty inline boxes, but these boxes still have margins, padding, borders and a line height, and thus influence these calculations just like elements with content.

Leading Leading (denoted L) is extra space added above and below text when computing line-height (denoted lh). It is equal to the computed line-height minus the *ascent* and *descent* of the text.

$$\mathsf{L} = \mathsf{lh} - (\mathsf{A} + \mathsf{D})$$

The ascent (denoted A) and descent (denoted D) of a text box are properties of the font of that text box. Ascent is the maximum height above the baseline, and descent the maximum depth below the baseline, for all characters of that font. See Figure 4 for a visual representation of ascent, descent, and leading. Note that if A + D < lh the leading is negative, which is allowed.

[13] By moving the dropdown menu 99 999 pixels left of the screen.

C.2 Margin Collapsing

The CSS 2.1 specification defines the rules for margin collapsing in section 8.3.1 [54]:

> In CSS, the *adjoining* margins of two or more boxes (which might or might not be siblings) can combine to form a single margin. Margins that combine this way are said to collapse, and the resulting combined margin is called a collapsed margin. Two margins are adjoining if and only if:
>
> - both belong to in-flow block-level boxes that participate in the same block formatting context
> - no line boxes, no clearance, no padding and no border separate them (certain zero-height line boxes (see 9.4.2) are ignored for this purpose)
> - both belong to vertically-adjacent box edges, i.e. form one of the following pairs:
>   - top margin of a box and top margin of its first in-flow child
>   - bottom margin of box and top margin of its next in-flow following sibling
>   - bottom margin of a last in-flow child and bottom margin of its parent if the parent has 'auto' computed height
>   - top and bottom margins of a box that does not establish a new block formatting context and that has zero computed 'min-height', zero or 'auto' computed 'height', and no in-flow children

However, not all margins collapse. Here is a summary of margins that are specified not to collapse in CSS 2.1:

> Horizontal margins never collapse. Margins of the root element's box do not collapse. If the top and bottom margins of an element with clearance are adjoining, its margins collapse with the adjoining margins of following siblings but that resulting margin does not collapse with the bottom margin of the parent block.

C.3 Floating Layout

The CSS 2.1 specification defines the rules for floating layout in section 9.5.1 [54]:

1. The left outer edge of a left-floating box may not be to the left of the left edge of its containing block. An analogous rule holds for right-floating elements.
2. If the current box is left-floating, and there are any left-floating boxes generated by elements earlier in the source document, then for each such earlier box, either the left outer edge of the current box must be to the right of the right outer edge of the earlier box, or its top must be lower than the bottom of the earlier box. Analogous rules hold for right-floating boxes.
3. The right outer edge of a left-floating box may not be to the right of the left outer edge of any right-floating box that is next to it. Analogous rules hold for right-floating elements.
4. A floating box's outer top may not be higher than the top of its containing block. When the float occurs between two collapsing margins, the float is positioned as if it had an otherwise empty anonymous block parent taking part in the flow. The position of such a parent is defined by the rules in the section on margin collapsing.
5. The outer top of a floating box may not be higher than the outer top of any block or floated box generated by an element earlier in the source document.
6. The outer top of an element's floating box may not be higher than the top of any line-box containing a box generated by an element earlier in the source document.
7. A left-floating box that has another left-floating box to its left may not have its right outer edge to the right of its containing block's right edge. (Loosely: a left float may not stick out at the right edge, unless it is already as far to the left as possible.) An analogous rule holds for right-floating elements.
8. A floating box must be placed as high as possible.
9. A left-floating box must be put as far to the left as possible, a right-floating box as far to the right as possible. A higher position is preferred over one that is further to the left/right.