Automated Reasoning for Web Page Layout

Pavel Panchekha

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Zachary Tatlock, Chair

Michael D. Ernst, Chair

Shoaib Kamil

Program Authorized to Offer Degree:
Computer Science and Engineering

©Copyright 2019
Pavel Panchekha

University of Washington

Abstract

Automated Reasoning for Web Page Layout

Pavel Panchekha

Co-Chairs of the Supervisory Committee:
Assistant Professor Zachary Tatlock
Computer Science and Engineering

Professor Michael D. Ernst
Computer Science and Engineering

Web pages define their appearance using Cascading Style Sheets. However, the CSS language's quirks and subtleties make it difficult for designers to write, debug, and test CSS to implement their designs. Additionally, web pages can be laid out by different browsers, at varying screen sizes, and by users with different font or color preferences. Understanding all the ways a web page can be laid out, and ensuring that all these layouts are correct, is a persistent challenge. Yet meeting this challenge is essential to ensure that a web page is accessible, usable, and attractive for all users. Tools to verify such web page layout properties across a range of possible layouts would lighten the testing burden for web developers.

We develop a formal verification paradigm for layout properties to support such tools. To formally verify a layout property, that property is first expressed in Visual Logic, a concise language for describing geometric properties of web page layouts. Visual Logic properties are true or false as a consequence of how a browser lays out a given page; we formalize a substantial fragment of browser layout in the Cassius framework. This formalization allows automated reasoning about CSS files and thus enables new tools that reason about the possible layouts of a web page. One such tool, VizAssert, automatically verifies that layout properties, expressed in Visual Logic, are true across a range of screen size or user config-

urations. Since VizAssert operates on whole web pages, it is difficult to scale to large web pages. However, large web pages are frequently composed of smaller, independent pieces. Reasoning about large pages should leverage this independence; to this end, the Troika proof assistant allows decomposing a large web page into many components and verifying those components independently.

These tools provide a demonstrate formal verification for layout properties and sketch a path toward provably correct web design.

TABLE OF CONTENTS

Page

Chapter 1: Introduction . . . 1
- 1.1 Formalizing Visual Properties . . . 3
- 1.2 Formalizing Browser Rendering . . . 4
- 1.3 Automatic Verification . . . 5
- 1.4 Modular Reasoning . . . 6
- 1.5 Results . . . 7

Chapter 2: Background on Web Browsers . . . 9
- 2.1 HTML, CSS, and JavaScript . . . 9
- 2.2 Web Browsers . . . 10

Chapter 3: Worked Example: Verifying a Web Page Layout . . . 14
- 3.1 Formalizing Accessibility Guidelines . . . 14
- 3.2 Verifying Assertions . . . 16
- 3.3 Writing a Modular Layout Proof . . . 17
- 3.4 Reading this Dissertation . . . 23

Chapter 4: Formalizing Visual Properties . . . 24
- 4.1 Visual Logic . . . 24
- 4.2 Semantics of Visual Logic . . . 29
- 4.3 Case Studies . . . 30

Chapter 5: Formalizing Browser Layout . . . 41
- 5.1 The Cassius Framework . . . 41
- 5.2 CSS Semantics . . . 45
- 5.3 Floating Layout . . . 53

Chapter 6: Automating Verification 60
6.1 How VizAssert uses Cassius 60
6.2 SMT Encoding 62
6.3 Finitization Reductions 65
6.4 Reasoning about Floating Layout 71

Chapter 7: Modular Reasoning 76
7.1 Modular Layout Proofs 76
7.2 Component Specifications 79
7.3 The Troika Proof Assistant 85

Chapter 8: Related Work 89
8.1 Accessibility of Web Pages 89
8.2 Formalizing Web Page Layout 93
8.3 Web Developer Tools 100
8.4 Formal Methods 104

Chapter 9: Evaluation 107
9.1 CSS Working Group Unit Tests 107
9.2 Free Website Templates 113
9.3 Joel on Software 125

Chapter 10: Conclusion 132
10.1 CSS Features outside the Cassius Formalization 132
10.2 Extensions to Cassius 133
10.3 Debugging and Synthesizing CSS 134
10.4 Conclusions 135

Bibliography 137

ACKNOWLEDGMENTS

I would like to thank the many people who contributed to this work: my advisors Michael D. Ernst and Zachary Tatlock; my collaborators Shoaib Kamil and Emina Torlak; my colleague Adam Geller; my committee members James Fogarty and Andy J. Ko; the many members of the UW CSE PLSE lab who read drafts and offered feedback; and the PLDI and OOPSLA reviewers and shepherds.

I would also like to thank every who supported my PhD, especially: my parents Anna and Alexey Panchekha; Gerald J. Sussman, who encouraged me to pursue research; and my friends Mark Velednitsky, Katherine Fang, and Edward Zhang for supporting me in difficult times.

Finally, I would like to thank my colleagues and collaborators on the Herbie and FPBench projects: Alex Sanchez-Stern, David Thien, Bill Zorn, and Chen Qiu. Herbie is not mentioned in this dissertation but formed an important strand of my graduate work.

Chapter 1

INTRODUCTION

There are over 7.4 million adults with a vision disability in the USA alone [Nat16]. Many users with a vision disability use screen-readers or other accessibility technologies to browse the web. Other users have sensorimotor disabilities that restrict their use of input devices. Graphical interfaces can support these users by ensuring that graphical interfaces have large, high-contrast text and by adding annotations that enable screen-reader use. GUI designers can improve usability for these users by following guidelines and best practices, such as those from the Department of Justice [US 17a] or the W3C's Web Content Accessibility Guidelines [W3C08]. For some applications, adherence to these guidelines is mandated by the European Web and Mobile Accessibility Directive, by the Americans with Disabilities Act, or by the Basic IT Law of Japan [US 10, Eur16, Die00]. The guidelines should be satisfied regardless of screen size, user preferences, or device. Usability guidelines are just as broadly applicable, equally important to verify, and similar in flavor: accessibility might require buttons large enough for users with sensorimotor disabilities while usability requires buttons large enough for users on a mobile device. Even basic design guidelines (such as spacing and alignment) can be worth verifying, given the importance users place on attractive and easy-to-use interfaces.

However, on many pages these guidelines are undermined by layout bugs: for certain screen sizes or user preferences the web page will render in a way that violates the guideline, despite the web developer's best efforts to ensure conformance. Such layout bugs can make pages unusable for of users with vision disabilities. Developers are eager to fix accessibility issues, but discovering those issues is difficult [SM00, MFT05] due to the variety of browsers, operating systems, and devices, in addition to user preferences for fonts and

colors [HBGLB15, MFT05]. Exhaustively testing pages across the infinitely-many possible combinations of these parameters is impossible.

Accessibility and usability guidelines must be satisfied on all of the infinitely many combinations of *browser 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 [Dav12, FS06, CYM11]. These techniques only consider a particular subset of browser parameters and are not sufficient to ensure accessible applications [MFT05, SM00, IC02, RGSB00]. To verify accessibility requires checking all possible combinations of browser parameters. Note that accessibility and usability errors cause neither crashes nor exceptions. This lack of notification makes developers more likely to overlook accessibility failures, and makes verification especially valuable.

In this work, we introduce formal layout verification, a new paradigm for ensuring that web pages satisfy accessibility and usability guidelines:

1. To formally verify an accessibility or usability guideline, a compliance specialist first encodes that guideline as a formal specification in *Visual Logic*, a specialized logic for describing visual properties of web page layouts.

2. This formal specification is given a precise meaning by *Cassius*, a formalization of the browser rendering algorithm, which precisely describes the possible layouts of a given page across a range of browser parameters. Cassius implements the core of CSS 2.1 using the quantifier-free theory of linear real arithmetic (QF-LRA).

3. The formal specification can now be verified using *VizAssert*, which takes as input a web page, ranges for each browser parameter, and a formal specification, and either certifies that the specification is satisfied for all browser parameters in those ranges, or provides a counterexample for which the specification is violated.

4. The formal verification can be scaled to large web pages using the *Troika* proof assistant, which allows a verification engineer to decompose a large page into multiple components, verify each component, and assemble the component proofs into a proof for a whole-page property.

To demonstrate the practicality of this approach, this thesis describes usable implementations of Visual Logic, VizAssert, Cassius, and Troika. Visual logic is used to formalize a collection of 14 accessibility, usability, and design guidelines. The Cassius framework is tested on 3651 browser conformance tests, demonstrating that Cassius semantics match the implicit semantics of Mozilla Firefox. VizAssert is then used to verify visual logic assertions on 51 professionally-designed web templates, verifying 349 assertions and finding 38 bugs. Finally, Troika is demonstrated on a case study 11× larger than the template pages, showing that modular verification produces speed ups of 11–1027 ×.

1.1 Formalizing Visual Properties

Web developers wish to ensure that their web pages satisfy high-level, English-language guidelines. However, testing whether informal, English-language guidelines are satisfied on a particular web page fundamentally requires human interpretation. In the formal verification paradigm, guidelines are instead formalized in an unambiguous, machine-readable language. Such a language must allow users to identify particular elements on the page (potentially quantifying over several elements at a time) and to describe the geometric relations between these elements. Ideally, the language would reflect the existing architecture of the web, using common standards like CSS selectors for identifying page elements and exposing aspects of the CSS box model like margins, borders, and padding.

Visual Logic is such a language. Visual Logic is a compact mathematical logic with operations for traversing a page's tree structure and for stating geometric properties on its elements. Visual Logic can also be easily extended to a particular domain (such as web pages) by introducing constructs such as CSS selectors and the CSS box model. These extensions

allow Visual Logic to formalize common accessibility, usability, and design guidelines. We identify two key challenges in formalizing guidelines where human interpretation and intuition is requried: identifying the page elements that a particular guideline is relevant to; and formalizing high-level design principles such as visual importance, alignment, and spacing.

1.2 Formalizing Browser Rendering

To determine whether a visual logic assertion is true or false for a particular web page requires understanding how that page will be laid out by various browsers at a range of browser parameters (such as screen size, available fonts, and user preferred font size). This understanding is provided by Cassius, an extensive formalization of the browser rendering algorithm for a core fragment of CSS 2.1, including complex features such as line height, margin collapsing, and floating layout. Furthermore, Cassius formalizes this algorithm in the theory of quantifier-free linear real arithmetic (QF-LRA), enabling reasoning about CSS using off-the-shelf Satisfiability Modulo Theories (SMT) solvers.

Cassius focuses on the core aspects of the CSS semantics: cascading stylesheets, generating boxes for HTML elements, and laying out the resulting boxes. Each of these steps is modeled in detail, from high-level structure such as multiple layout modes to the details of shrink-to-fit, positioning, and text layout. In essence, Cassius is a feature-rich, declarative implementation of a browser layout engine, consisting of a set of QF-LRA constraints that relate browser inputs (the CSS stylesheet and the HTML document) and parameters (such as browser size and user preferences) to the browser output (the resulting box-based layout).

Two key insights enable Cassius. The first is identifying a fragment of CSS that is both expressive and amenable to layout with a *non-deterministic incremental algorithm*. Such an algorithm is naturally expressible in QF-LRA. The second is developing a technique, based on *auxiliary uninterpreted functions*, for efficiently encoding an arbitrary run of this algorithm in QF-LRA. Auxiliary uninterpreted functions allow reducing the size of the QF-LRA query, ensuring that is linear in the size of the web page. Together, these two innovations enable Cassius to reduce a given layout problem to an efficiently-solvable quantifier-free formula.

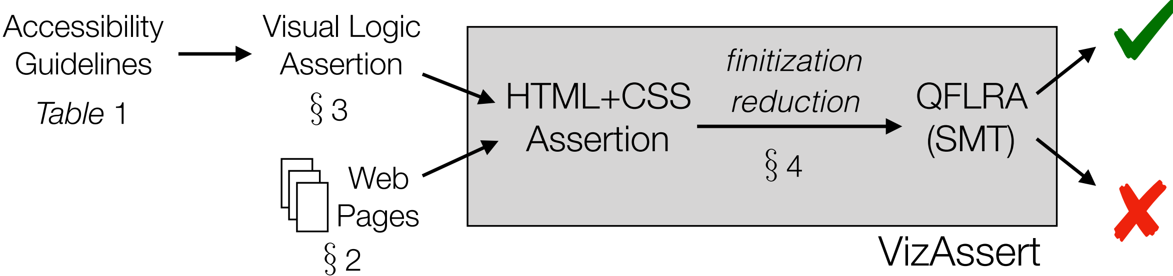


Figure 1.1: VizAssert ensures that web pages satisfy accessibility guidelines. VizAssert transforms those assertions, expressed in visual logic, into properties of the HTML and CSS source of the web page, then encodes those properties to formulas in the quantifier-free theory of linear real arithmetic.

To ensure its correctness, Cassius is thoroughly compared to Mozilla Firefox's layout engine, providing confidence that Cassius's formalization agrees with major browsers.

1.3 Automatic Verification

Given a formal specification in Visual Logic and a web page whose possible layouts are given by Cassius, the formal specification can be soundly verified by searching for a possible layout where the formal specification is false. This task is automated by the VizAssert tool: VizAssert uses an SMT solver to automatically search the set of possible layouts, outputing "verified" if no browser parameters can be found that causes a violation of the assertion. The assertion then provably holds for all considered browser parameters. On the other hand, if the search succeeds, VizAssert outputs "violated" along with a *counterexample* that identifies a set of boxes on the page and values for the browser parameters for which the assertion is violated. The developer can examine the counterexample in a browser to confirm the error and then either modify the page to fix it or modify the specification to allow it. Figure 1.1 summarizes this process.

Verifying real web pages involves a key technical innovation: novel *finitization reductions* for reasoning about arbitrary-size sets. The CSS standard describes several ubiquitous aspects of CSS in terms of operations on arbitrary-sized sets, among them line height computation, margin collapsing, and floating layout. However, such arbitrary-sized sets are difficult for an SMT solver to reason about, foiling automated verification. *Finitization reductions* replace the arbitrary-size sets discussed by the standard with compact, finite-size representations which are more amenable to mechanical reasoning. This produces an efficiently-solvable SMT query, allowing VizAssert to use Z3 [DMB08] to automatically search for erroneous layouts or to certify that no such layouts exist.

1.4 Modular Reasoning

Monolithic analyses like VizAssert have, however, several disadvantages. They do not scale to large pages with many elements and cannot parallelize verification of a webpage. Even for the modestly-sized pages they can handle, current analyses are unnecessarily slow because every run requires re-checking the entire page from scratch, even for small changes like typo fixes. Furthermore, the diversity of tools and approaches with varying strengths and weaknesses suggests that no single tool is best for verifying every web page.

In other domains, modular verification enables scaling [CCG+03, DMS+09, JSS+15, App16] by using abstraction to break the verification problem into smaller sub-problems. But previous modularity techniques do not apply directly to web page layout, because web pages lack clear computational units like functions that define module boundaries. Web page layout is a constraint problem: the layout of each component depends on, and potentially affects, all other components on the page—both those before *and* after it. The shared "state" through which web page elements interact is the geometry of the elements' graphical layout. In short, the scalability of existing techniques is limited because they do not summarize component behavior in a composable and independently-checkable manner that supports modular reasoning, and techniques for introducing modularity from other domains do not apply directly to web page layout.

Modular layout proofs scale layout verification to larger pages and sites. A modular layout proof partitions a web page into *components*: contiguous fragments of the page's HTML structure paired with symbolic summaries of its CSS style. Components are the units of web page modularity. A *component specification* summarizes a component's possible layouts. These summaries can be used to prove whole-page properties without unscalable reasoning about the browser layout algorithm. Components may affect each other's layout; component specifications constrain such interactions using rely/guarantee-style preconditions [Sta85], allowing each component specification to be verified independently.

Verifying large pages in small pieces has three benefits. (1) Unscalable verification techniques can be used on each small piece in parallel, enabling verification of pages that are too large to verify monolithically. (2) Verified components can be cached and reused without re-verification. (3) Different verification tools, such as model checkers and SMT-based verifiers, can verify different component specifications. Scalable algorithms for checking the correctness of modular layout proofs capitalize on these benefits, allowing verification of pages an order of magnitude larger than previous approaches.

This approach to modular web page verification is implemented in a new proof assistant, named Troika. A proof author uses Troika's tactic language to decompose a web page into components and define specifications for those components. Troika then automatically verifies the component specifications and checks the modular layout proof. Troika integrates multiple tools for checking component specifications, including a whole-page SMT-based verifier, a new specialized verifier for component specifications, a model checker, and a random testing tool. Troika allows proof authors to apply each tool where it produces the best results, and to smoothly transition from testing to verification. Troika also speeds up verification by parallelizing component verifications and caching results.

1.5 Results

To ensure that Cassius correctly formalizes the browser layout algorithm, we compare Cassius to Firefox on a collection of 3651 CSS conformance tests covering the subset of CSS 2.1

formalized in Cassius. Cassius matches Firefox's layout in almost all cases, and the few differences identify known bugs in Firefox.

To demonstrate that VizAssert is a powerful and efficient verification tool, we consider 14 accessibility guidelines and best practices, formalize each accessibility guideline in visual logic, and check them on a collection of 51 professionally designed web pages. Section 4.1 discusses the adaptations and choices necessary to turn informal guidelines and best practices into formal assertions. Of the assertions, 6 are page-specific, demonstrating that users can write custom assertions to verify domain-specific properties of their web pages. More broadly, the particular set is less important than the fact that VizAssert is general and extensible. In all, there are 414 combinations of web page and assertion. Of these, VizAssert found 38 assertion violations, and issued 21 false positive warnings. We write Troika proofs for 8 of the combinations; the Troika proofs are 1.8–64 × faster than VizAssert.

Finally, Troika allows scaling verification to large web pages. We perform a case study with Troika on the "Joel on Software" blog [Spo18], and prove two key accessibility properties; Troika checks the proof 11–1027 × faster than the monolithic VizAssert tool. The proof is short (36 lines) and reusable on websites with similar style; Troika's support for multiple verification tools was critical. Troika caches and reuses component verifications across modifications to a single page and across multiple pages (e.g., different blog posts) on one site.

Chapter 2
BACKGROUND ON WEB BROWSERS

To show web pages to end users, web browsers interpret HTML and CSS source code. HTML defines the content of a web page, while CSS defines its appearance. Browsers use this information to *lay out* the page while consulting *browser parameters* such as browser window width and height.

2.1 HTML, CSS, and JavaScript

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 browser to render bold text.

CSS A CSS stylesheet contains a list of *rules*. Each rule is guarded by a *selector*, which restricts the rule to apply only to specific elements, such as only to paragraphs that are children of the article text. Some selectors also include *media queries*, which turn rules on or off based on browser parameters like screen size. The body of a rule is a set of declarations—pairs of *properties* and *values*. The following CSS file contains two rules:

```
@media (min-width: 500px) { body { margin: .5em; } }
b { font-weight: bold; }
```

The first rule selects elements with the `body` tag and sets their `margin` property; it is guarded by a media query, so that the rule only applies if the screen size is greater than or equal to 500 pixels wide. The second rule selects all elements with the `b` tag and sets their `font-weight` property; it has no media query guarding it, so it always applies. CSS also has selectors for selecting elements by their relationship to other elements or by attributes attached to those elements. For every possible property, a browser gathers and ranks all the rules that set that property, and every element gets its value for that property from the highest-ranked rule that applies to it.

JavaScript Interactive web pages also contain JavaScript code. JavaScript does not directly affect layout; it only modifies the run-time representation of the page HTML (by modifying the Document Object Model) and CSS (by modifying inline styles). The browser then uses the layout algorithm to display the modified page.

2.2 Web Browsers

A web browser has many jobs. It must:

1. draw an attractive and convenient user interface (usually called browser chrome) to allow users to easily navigate to various web pages;

2. fetch web pages (including HTML, CSS, and JavaScript) over the network (over many different protocols);

3. lay out web pages using the HTML and CSS fetched, determining where various elements on the page go;

4. render text with complex typography (including multiple fonts, multiple scripts, mixed-direction writing, sub- and super-scripts, ligatures, kerning, and hyphenation);

5. draw the web page (including text) to the screen, usually using GPU acceleration to speed this process up;

6. run plug-ins like Flash and Java, as well as add-ons that modify the browser's chrome or its functionality;

7. support a sandboxed JavaScript execution environment (including just in time compilation, multiple threads, client-side databases, 3D graphics, and access to hardware features like cameras, video encoders, and GPS modules);

8. and enforce a security policy across plug-ins, JavaScript, and add-ons, including eliminating leaks of browser history or cookies.

Given this wealth of features, it is no surprise that browser implementations are complex. Mozilla Firefox, for example, contains over 19 million lines of code, the plurality in C++ [Blab], while Chromium (the open source version of Google Chrome) contains over 25 million, also mainly in C++ [Blaa].

This thesis focuses on a single important task web browsers fulfill: laying out web pages. This task combines the HTML and CSS to produce a *box tree* containing all of the page's visual elements and their sizes, positions, and colors (see Figure 2.1). Generally, the HTML tree and layout tree have similar structure, but a single element can produce zero boxes (for invisible elements) or multiple boxes (for list bullets). *Standards documents* from the World Wide Web Consortium [W3C11] specify the browser layout algorithm. These standards are informal English language documents, and ambiguities are often resolved by consensus among major browser vendors instead of directly in the standard. These standards evolve over time as browsers develop new features. Though browser conformance is poor for recently-standardized features, all modern browsers behave similarly for the core CSS features discussed in this work.

Laying out a web page requires consulting *browser parameters*, such the height and width of the browser window, which affect the layout. Some browser parameters are obscure: how

a page is laid out may depend on user preferences like the user's preferred fonts and font sizes; operating system options that determine the size of buttons, scroll bars, and drop-downs; network quirks, like which third-party fonts can be downloaded by the browser; or even browser quirks like different hyphenation algorithms. Browser parameters are added and removed as browser implementations evolve; for example, since 2018 Apple Safari has supported a "dark mode", which users can switch on and which changes the colors of page elements. Some of these parameters affect media queries (such as browser window width and height, or dark mode) while others only affect web page layout implicitly; for example, buttons are larger on Windows than on macOS, so web pages with buttons move elements around to make room for the buttons on different platforms.

This dissertation describes a formalization of the browser layout algorithm, with particular focus on its dependence on browser parameters (since the ultimate goal of this work is verification of a web page across a range of possible browser parameters). In other words: this dissertation describes an implementation of browser layout in the restricted language of first-order logic. Luckily, this implementation is not burdened by many of the considerations that make major browser implementation complex. Major browsers implement *incremental* layout engines, which can efficiently lay out a page after minor changes (for example after JavaScript changes the page). Incremental layout engines track additional dependency information to avoid recomputing values that didn't change. Furthermore, major layout engines must *sequence* the computation of sizes, positions, and colors to ensure that each value's dependencies are available when that value is computed. Sequencing these computations can be complex, since different values have different dependencies; for example, widths are computed in a top-down traversal, heights are computed bottom-up, and floats are computed in-order. Neither incrementality nor sequencing are important for formalizing the browser layout algorithm.

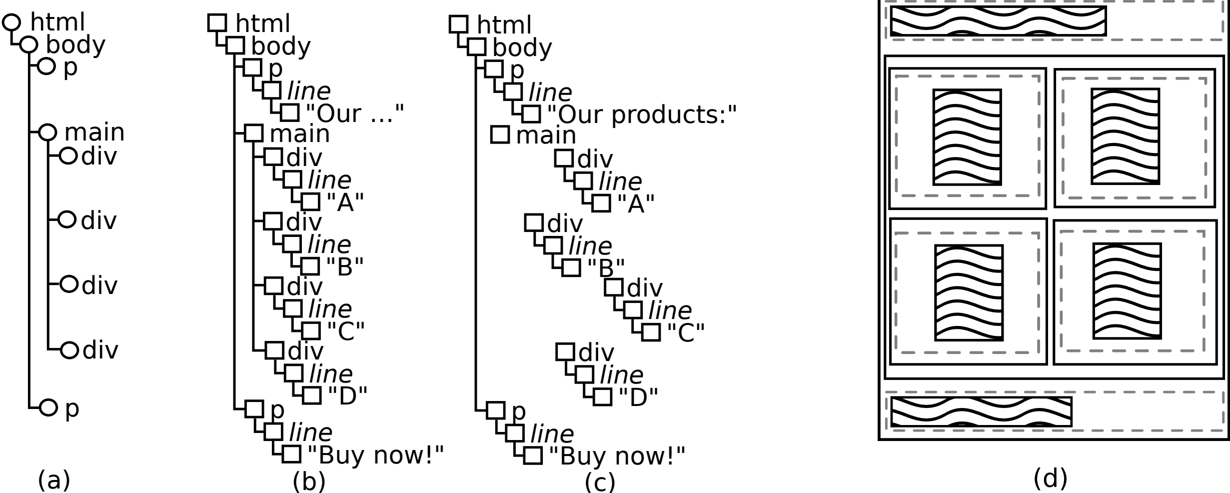


Figure 2.1: Layout converts HTML into a tree of elements (2.1a). The tree of elements generates a tree of boxes (2.1b), which includes block, line, and text boxes, and this tree is broken into flow trees (2.1c). A position and size is then computed for every box, resulting in the layout (2.1d).

Chapter 3
WORKED EXAMPLE: VERIFYING A WEB PAGE LAYOUT

This section demonstrates formal verification of web page accessibility properties by stepping through a hypothetical verification that all links are in a scrolling-accessible location on the screen[1] for the "yoga studio" web page inset in Figure 3.1. This example covers three major components to the formal verification approach developed in this dissertation: Visual Logic, which is used to formalize the link-scrollable assertion; VizAssert, which is used to automatically verify the property; and Troika, which is used to develop a modular and reusable proof of the same property. The Cassius formalization of browser layout plays a background role, allowing VizAssert and Troika to carry out their verification steps. Each of these four tools is described in later chapters; this chapter demonstrates how the pieces fit together into the larger vision of web page layout verification.

3.1 Formalizing Accessibility Guidelines

Users with difficulty seeing often increase the default font size, but some pages are not functional when this setting is changed. For example, on some pages, the interactive elements of the page go off-screen and cannot be accessed when the font size is changed, because the developer failed to test the page on multiple font sizes. This notion can be expressed precisely in Visual Logic:

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

This specification demands that all interactive elements be scrolling-accessible; both those predicates still need to be defined. We found that links were the most common interactive

[1] For a browser window 800–1920 pixels wide, and any default font size between 16 and 32 pixels.

```
1  define scrollable(b) = b.left ≥ 0 ∧ b.top ≥ 0
2  theorem links-scrollable = ∀b, b ∈ $(a) ⟹ scrollable(b)
3  page yoga = load yoga/index.html with
4    browser.width ∈ [800, 1920]
5    browser.height ∈ [600, 1280]
6    font.size ∈ [16, 32]
7  proof of links-scrollable for yoga
     # Subdivide the page into components
8    component head = $(#header)
9    component body = $(#body)
10   component foot = $(#footer)
     # Component specifications for each component
11   for all c ∈ 𝒞 assert ∀b, scrollable(c) ∧ b ∈ $(a) ⟹ scrollable(b) by component-smt
12   for root assert scrollable(head) ∧ scrollable(body) ∧ scrollable(foot) by component-smt
     # Preconditions for the footer width
13   for foot require foot.width ≥ 200
14   for root assert foot.width ≥ 200 by component-smt
     # Preconditions for floating boxes
15   for foot require no-floats-enter(foot)
16   for all c ∈ 𝒞 assert no-floats-enter(c) ⟹ no-floats-exit(c) by component-smt
17   for root assert no-floats-enter(root) by component-smt
18   for root assert float-flow-in(root, header) by component-smt
19   for root assert float-flow-across(header, body, footer) by component-smt
20   for all c ∈ 𝒞 assert non-negative-margins(c) by component-smt
21 qed
```

Figure 3.1: A complete proof that all links on the yoga page are in a scrolling-accessible location. Section 7.3.1 gives the semantics of the tactic language. The inset screenshot shows the yoga page and its decomposition into components (white-outlined rectangles).

elements:

$$\textbf{define}\ \mathsf{interactive}(b) = b \in \$(\mathsf{a})$$

An on-screen box must be either to the right or below the top-left corner of the screen, which is the page origin (on a web page y-values increase down the page, with the origin at the top):

$$\textbf{define}\ \mathsf{scrollable}(b) = b.\mathsf{left} \geq 0 \land b.\mathsf{top} \geq 0$$

Note that this property does not restrict boxes from being past the right or bottom edge of the screen, since users can still see such elements by scrolling.

Formal verification means checking this property for the set of all font sizes (as well as all other browser parameters) that the website intends to support. A common guideline is to support font sizes between 16 pixels, the default, and 32 pixels,[2] allowing a zoom of up to 200%; this guideline matches the default settings of the formal verification tools described in this dissertation.

3.2 Verifying Assertions

With the accessibility guideline formally stated, VizAssert can be used to verify that the guideline holds of the yoga studio web page. To do so, the yoga studio web page must be *captured*, transforming it into a syntax used by the Cassius framework (see Figure 5.2 for an example):

```
python capture/capture.py example/yoga.html --output yoga.cassius
```

VizAssert can now be invoked to verify the guideline on this page:

```
racket src/run.rkt assertion assertion.va links-scrollable \
       yoga.cassius doc-1
```

This invocation instructs VizAssert to perform a sequence of steps:

[2] 12 and 24 points.

1. Load the links-scrollable assertion from the file `assertion.va`;
2. Compile that visual logic formula to a formula in the quantifier-free logic of linear real arithmetic;
3. Load the doc-1 web page[3] from the file `yoga.cassius` captured earlier;
4. Use the Cassius framework to express the possible layouts of that page as a set of formulas also in quantifier-free linear real arithmetic;
5. Pass this large collection of QF-LRA formulas to the Z3 SMT solver, to determine whether the assertion is false of any possible layout;
6. If no, output "Verified";
7. If yes, extract from the solver's counterexample model a set of browser parameters and particular page elements that allow this possible layout to be reproduced in an ordinary browser.

For the yoga studio web page, which contains 36 HTML elements, 239 CSS rules, and uses 8 fonts, VizAssert executes these steps in 5.1 minutes, with the vast majority spent waiting for Z3 in step 5.

3.3 Writing a Modular Layout Proof

While VizAssert makes it possible to automatically verify that a web page satisfies and accessibility property, its monolithic approach means that it is impossible to reuse a verification after a small change to the web page (as would be common during incremental web development). Troika overcomes this limitation and enables web developers to verify a web page property *modularly*, that is, by decomposing the web page into independent components,

[3]Captured files can contain multiple pages, and doc-1 refers to the first one.

specifying facts about the layout of each component (which can be verified independently, using a mix of different verification tools), and proving the accessibility property from those component facts. Figure 3.2 illustrates the overall workflow.

A modular layout proof of a property Q for a web page p consists of two parts: a partitioning of the page p into *components* $c \in \mathcal{C}$, which are contiguous regions of the HTML tree with CSS style information; and per-component *specifications* P_c, which summarize relevant facts about the possible layouts of that component (and thus abstract away the intricacies of the browser layout algorithm). The modular layout proof establishes Q if each specification P_c is true of all possible layouts of the component c on the page p and the conjunction of the specifications P_c imply the whole-page property Q, a condition called *well-formedness*:

$$\left(\bigwedge_c P_c\right) \implies Q$$

The key innovation of modular layout proofs is that this implication must be true as a matter of pure logic, independent of the details of web page layout. This also allows each component specification to be established by different methods for reasoning about web page layout, even if those methods model web page layout differently.

For the yoga studio page, the property Q is the visual logic formula described above:

1 **definition** $\mathsf{scrollable}(b) = b.\mathsf{left} \geq 0 \wedge b.\mathsf{top} \geq 0$

2 **theorem** $\forall b, b \in \$(\mathsf{a}) \implies \mathsf{scrollable}(b)$

(The line numbers on the code examples in this section match the line numbers in Figure 3.1.)

With the layout property stated (Step 1 in Figure 3.2), the proof author can now develop a modular layout proof for it.

Defining Components The first part of a modular layout proof is a decomposition of the web page into components (Step 2 in Figure 3.2). The proof author chooses components by considering the structure and size of the page, balancing the competing concerns of human effort (writing fewer component specifications) and solver effort (reasoning about smaller

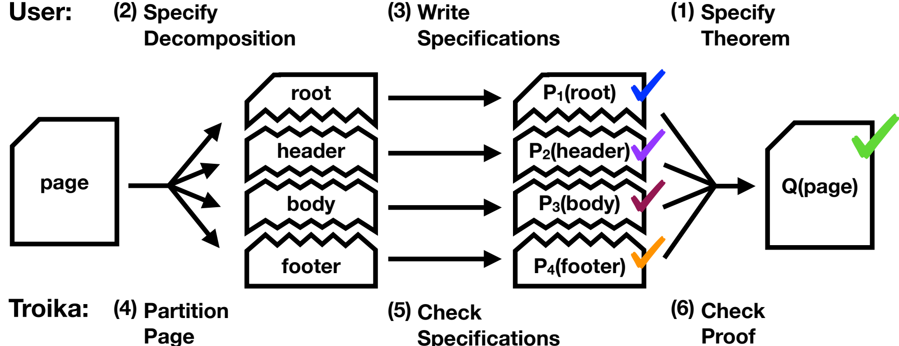


Figure 3.2: The Troika workflow. The proof author specifies a theorem Q to prove about a page, describes a decomposition of that page into components (here, the root, header, body, and footer), and writes specifications P_c for each component. Troika uses the decomposition to partition the page into components c, verifies each component's specification using various verification tools, and checks that Q follows from the P_c.

components).[4] For the yoga page, visual inspection suggests subdividing the page into a header, body, and footer (see the decomposition in Figure 3.1). The yoga page author already gave each of these components a CSS selector, which is standard practice.

```
8   component head = $(#header)
9   component body = $(#body)
10  component foot = $(#footer)
```

The root of the page, which the header, body, and footer fit into, also forms a component; in Troika this component is implicit and is named root.

Defining Component Specifications With the page decomposed into components, the second part of a modular layout proof assigns specifications to each component such that the overall theorem Q is implied by their conjunction (Step 3 in Figure 3.2). A Troika

[4] The case study in Section 9.3 is a good example of navigating this trade-off.

specification is expressed using the assert tactic,[5] written " **for all** $c \in S$ **assert** P_c **by** T", where S describes a set of components, P_c is a component specification, and T names the tool Troika should use to verify the assertion. A starting point for the component specifications in this page is to require the links in each component be scrollable:

$$\text{11} \quad \textbf{for all}\ c \in \mathcal{C}\ \textbf{assert}\ \forall b, b \in \$(\mathsf{a}) \implies \mathsf{scrollable}(b)\ \textbf{by}\ \mathsf{admit}$$

This assertion ranges over the set $\mathcal{C}$ of all components. The "admit" tool assumes that the specification holds without checking it. To complete the proof, we will later eliminate all uses of admit. (The line number is struck out because this line of code is not present in the final proof.)

Since every link is in some component, the component specifications assumed above imply the whole-page property. Furthermore, this conclusion does not depend on the details of browser layout; it is instead a matter of pure logic. These conditions mean that Troika can prove the whole-page property using the component specifications we have written: it takes 0.51 seconds (Step 6 in Figure 3.2).

3.3.1 Verifying Component Specifications

In a modular layout proof, each component specification may be verified using a different verification tool (Step 5 in Figure 3.2). These tools must ensure that every component specification P_c is true of all possible layouts of c on the web page p. Troika provides four tools that can check component specifications: "random-test", "model-check", "whole-page", and "component-smt".

Different components of a web page can affect each other's layout. The random-test, model-check, and whole-page tools handle these dependencies by reasoning about the full web page in order to verify a specification for just a part of that page. Reasoning about the whole page, however, is unscalable, so modular proofs commonly employ an alternative approach:

[5] Troika provides a language of commands for updating proof state (detailed in Section 7.3), similar to tactic languages in other proof assistants.

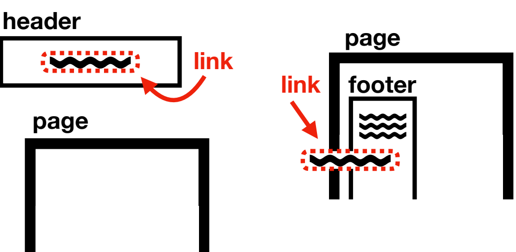


Figure 3.3: Two possible problems that the links-scrollable proof guards against using rely/guarantee-style preconditions. (1) If the header is placed off-screen, its links will also be off-screen. (2) If the footer is too narrow, its links will extend beyond its left edge.

constraining the effect of one component on another with rely/guarantee-style preconditions. We call a component specification *independently-true* if it constrains interactions with other components using preconditions sufficiently for the component to be verifiable independently of the rest of the page. component-smt is an efficient tool for checking such specifications.

To develop the preconditions needed by independently-true component specifications, a proof author writes **assert** tactics that use the component-smt tool. If the component specification is not independently-true, component-smt outputs a counterexample: a web page that contains the component and in which the component violates the specification. The proof author then weakens the specification by adding a precondition that rules out the counterexample. Each precondition must be established by an assertion in some other component. This process continues until the proof is verified.

Generic Specifications As an example, suppose that a proof author wishes to make the specification $\forall b, b \in \$(\mathsf{a}) \implies \mathsf{scrollable}(b)$, which was admitted above, independently-true in the header. The proof author would start by changing "by admit" to "by component":

11 **for all** $c \in \mathcal{C}$ **assert** $\forall b, b \in \$(\mathsf{a}) \implies \mathsf{scrollable}(b)$ **by** component

This specification is not independently-true, so Troika produces a counterexample, in which the header itself is placed offscreen (see Figure 3.3, left). To prevent this counterexample,

the proof author would add the precondition scrollable(head):

$$\text{11} \quad \textbf{for } \mathsf{head} \textbf{ assert } \forall b, \mathsf{scrollable}(\mathsf{head}) \land b \in \$(\mathsf{a}) \implies \mathsf{scrollable}(b) \textbf{ by } \mathsf{component\text{-}smt}$$

To ensure that the overall proof is still well-formed, this precondition must also be asserted in some other component:

$$\text{12} \quad \textbf{for } \mathsf{root} \textbf{ assert } \mathsf{scrollable}(\mathsf{head}) \textbf{ by } \mathsf{admit}$$

Of course, this "admit"ed assertion must later be proven using a sound component verification tool. If we were to use whole-page, the proof would require reasoning about the browser layout algorithm for the full page; component-smt will only reason about the small root component, which consists of just the `html` and `body` tags and abstracts away the header, body, and footer components.

The scrollable precondition reflects a general property of web design: elements are generally located inside their ancestors. The new component specification establishes the proof's strategy: prove that each component is scrollable (in the root component) and use that to prove that each link is scrollable. Since it captures a general property of web design, we call such a component specification *generic*.

Page-Specific Preconditions Other preconditions reflect the idiosyncrasies of a particular web page. Consider the yoga page's footer. If the footer is too narrow, links within it could be outside the scrollable area of the page even if the footer itself is not: see Figure 3.3, right. The footer thus requires a precondition establishing a minimum width; minimum and maximum widths are common preconditions. Such page-specific preconditions could be added to the footer's component specification like above, but to keep the proof organized, Troika also provides a "require" tactic that puts each precondition on its own line. This

makes it possible to keep each precondition close to the assertion that establishes it:[6]

```
13  for foot require foot.width ≥ 200
14  for root assert foot.width ≥ 200 by admit
```

A proof author can continue to add preconditions to make each component specification independently-true. Figure 3.1 shows the proof after several more iterations of this process. The proof has floating layout preconditions (described in Section 7.2.4) for the footer and root, and establishes them using floating layout and non-negative margins assertions on every component. Thanks to these preconditions, every component specification is independently-true and can be established using the component-smt tool.

3.4 Reading this Dissertation

This dissertation describes the major components of formal verification for web page layouts. First, Visual Logic language for describing layout properties is given syntax and semantics in Chapter 4. Then, the broad structure of the Cassius formalization of the browser layout algorithm is described in Chapter 5, including details of some of the more challenging aspects to the formalization. The VizAssert automatic verification tool is described in Chapter 6, and the specifics of encoding browser layout in quantifier-free linear real arithmetic are explained. Finally, the Troika proof assistant is introduced in Chapter 7, detailing both its tactic language and its verification algorithms. To evaluate formal verification for web page layouts, Chapter 9 uses three collections of web pages, ranging from conformance tests to a popular blog.

---

[6] The minimum width of 200 pixels was determined by inspecting the page source. Any value from 200 to 800 would lead to a correct proof, but the smaller value allows reuse in more contexts.

Chapter 4

FORMALIZING VISUAL PROPERTIES

Visual Logic is a limited first-order logic for describing the visual properties of user interfaces. As a formal language for specifying visual properties, Visual Logic forms the "user interface" for formal verification of web page layout properties. Visual Logic models a user interface as a tree of boxes, each of which represents a rectangle of content located on a two-dimensional screen. Visual Logic provides operations to traverse the box tree and allows the geometric properties of boxes to be defined by linear real arithmetic. In this form, Visual Logic could be used for web pages, operating system APIs, and major user interface frameworks; we adapt it to web pages by adding new constructs for selectors and the CSS box model. This chapter introduces Visual Logic, describes its semantics in detail, and describes the formalization of 14 web accessibility best practices in Visual Logic.

4.1 Visual Logic

Visual Logic expresses accessibility and usability assertions in formal and unambiguous terms.

4.1.1 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 4.1). These formulas describe properties of a tree of rectangular boxes annotated with sizes, positions, and colors. For such a tree, each box represents a rectangle of laid out content (at its annotated position with its annotated size) and the tree structure of these

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

$\langle cond \rangle$ ::= $\langle cond \rangle \wedge \langle cond \rangle \mid \neg \langle cond \rangle \mid \langle cond \rangle \vee \langle cond \rangle$
| $\langle real \rangle = \langle real \rangle \mid \langle real \rangle < \langle real \rangle \mid \langle real \rangle > \langle real \rangle$
| $\langle box \rangle = \langle box \rangle \mid \langle box \rangle$.type $= \langle type \rangle \mid \langle box \rangle$.whitespace

$\langle real \rangle$ ::= $\mathbb{R} \mid \langle real \rangle + \langle real \rangle \mid \langle real \rangle - \langle real \rangle \mid \mathbb{R} \times \langle real \rangle$
| $\langle box \rangle.\langle dir \rangle \mid \langle color \rangle.\mathrm{r} \mid \langle color \rangle.\mathrm{g} \mid \langle color \rangle.\mathrm{b}$

$\langle color \rangle$ ::= transparent | rgb($\langle real \rangle$, $\langle real \rangle$, $\langle real \rangle$)
| $\langle box \rangle$.fg | $\langle box \rangle$.bg | γ($\langle box \rangle$.fg) | γ($\langle box \rangle$.bg)

$\langle box \rangle$ ::= b_i | root | null | $\langle box \rangle$.ancestor($\langle cond^* \rangle$)
| $\langle box \rangle$.parent | $\langle box \rangle$.first-child | $\langle box \rangle$.last-child
| $\langle box \rangle$.next | $\langle box \rangle$.prev

$\langle type \rangle$ ::= window | inline | line | text | block

$\langle dir \rangle$ ::= top | right | bottom | left

Figure 4.1: Visual Logic, a language for formally describing visual layout properties. All quantifiers are over the set $\mathcal{B}$ of boxes in a layout.

boxes is assumed to correspond to some semantic organization.[1]

Most Visual Logic constructs are straightforward (and described in more detail in Section 4.2). Some domain-specific constructs include:

- Assertions are universally quantified over boxes "b_i".
- "$\langle box \rangle$.type = $\langle type \rangle$" checks the type of a box, which defines the sort of content the box contains.
- "$\langle box \rangle$.whitespace" determines whether a text box contains only whitespace or also contains other text.[2]
- The "$\langle box \rangle.\langle dir \rangle$" construct expresses the position of the box's edges (and thus the box's position and size).
- The γ function gamma-corrects RGB colors.[3]
- 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.

1. Since SMT solvers reason efficiently 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.

[1]The tree structure need not correspond to containment for the rectangles corresponding to each box. For example, siblings or even unrelated boxes are allowed to visually overlap.

[2]Such boxes are easy to accidentally create in GUI frameworks that allow mixing text and non-text content, such as on the web.

[3]See Section 4.2 for more details on gamma correction and why it is important for accessibility, usability, and design.

3. SMT solvers cannot reason about recursive properties, so Visual Logic provides the $\mathsf{ancestor}$ function, which represents the closest ancestor of a box that satisfies a conditional expression in one variable (written "?").

If no ancestor satisfies the $\mathsf{ancestor}$ 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,

$$\mathsf{null}.\mathsf{ancestor}(\mathcal{C}) = \mathsf{null}$$

$$b.\mathsf{ancestor}(\mathcal{C}) = b \ \textbf{if} \ \mathcal{C}[?/b] \ \textbf{else} \ b.\mathsf{parent}.\mathsf{ancestor}(\mathcal{C})$$

Evaluating a Visual Logic assertion on a concrete layout is straightforward, though verification—that is, searching the infinite set of all possible layouts for a counterexample to the assertion—is more challenging.

4.1.2 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 [Fow17], Cocoa [App17c], or Android [Pro17]. Adapting Visual Logic to one of these platforms involves adding platform-specific constructs to Visual Logic. Since this work is focused 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 [W3C11]:

$$\langle cond \rangle ::= \ \cdots \ | \ \langle box \rangle \in \$(\langle selector \rangle)$$

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 $\mathsf{android{:}id}$s 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.

To make writing Visual Logic easier, the following common abbreviations are used:

1. $b.\mathsf{width}[e] \equiv b.\mathsf{right}[e] - b.\mathsf{left}[e]$

2. $b.\mathsf{height}[e] \equiv b.\mathsf{bottom}[e] - b.\mathsf{top}[e]$

3. $x \leq y \equiv x < y \vee x = y$

4. $x \geq y \equiv x > y \vee x = y$

5. $x \neq y \equiv \neg(x = y)$

6. $P(x_{\{a,b\}}) \equiv P(x_a) \wedge P(x_b)$

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

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

where the calligraphic letters such as $\mathcal{X}$ represent Visual Logic expressions and square brackets such as $\mathcal{E}[\mathcal{C}]$ represent the context of an expression.

4.2 Semantics of Visual Logic

Visual Logic describes properties of *box trees*. A box tree is a 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 tuples 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,[4] $\langle color \rangle$ terms denote to triples of real numbers or the special `transparent` value, and $\langle box \rangle$ terms denote to nodes in the box tree. These nodes are assumed to have child, sibling, and parent pointers; box 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 is standard. 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.

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 layout algorithm; in assertions, the `text` type identifies 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 Visual Logic implementation does not enforce this requirement, but most uses of non-constants on both sides of a multiplication cause the

---

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

underlying solver to report `unknown`, which is then displayed 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 gamma-corrected (using the "γ" function), 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 amount of light emitted from the red, green, and blue diodes on the computer's display.[5] 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, implementations of Visual Logic must precompute gamma correction for all colors mentioned in the CSS file, which includes the foreground and background of any box. This approach is why Visual Logic does not allow gamma-correcting arbitrary colors constructed with the `rgb` construct.

4.3 Case Studies

This section demonstrates the expressiveness of Visual Logic via 14 examples. Each example starts from a guideline from Table 4.1, adapts it to web pages from the Free Website Templates suite (Section 9.2) and expresses that adapted guideline as an assertion in Visual Logic. Of the 14 guidelines, 8 were formalized so that the same assertion was meaningful on a large set of web pages, while 6 are formalized in a way that is specific to a single page. Section 4.3.1 describes the general-purpose assertions, while Section 4.3.2 covers page-specific assertions.

Common Tasks in Guideline Formalization Two broad challenges must be overcome to translate informal, English-language guidelines to assertions in Visual Logic: precisely

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

Table 4.1: Best-practice accessibility and usability guidelines that we formalized in Visual Logic and evaluated on professionally designed web pages (Section 9.2).

#	Description	Source
General assertions, applicable to many web pages		
1	Text is at least 14px tall	[W3C08]
2	Text can be resized by up to 200%	[W3C08]
3	Lines are no more than 80 characters	[W3C08]
4	Elements for screen-reader users are off-screen	[Pea17, Moz17]
5	The page does not require horizontally scrolling	[Cer11]
6	Headings form a visual hierarchy	[Pam14]
7	Text does not overlap	[App17b]
8	Lines are appropriately spaced	[But10]
Specific assertions, applicable to a single web page		
9	Text has sufficient contrast	[W3C08, App17a]
10	Text does not overlap image	[W3C08, App17a]
11	Dropdown menus are hidden when not selected	[W3S17]
12	Columns are vertically aligned	[But10]
13	Full link text is visible	[App17b]
14	Main button is big enough	[T12]

identifying the page elements discussed by the guideline and translating high-level design concepts to concrete visual properties. First, accessibility guidelines commonly reference general classes of visual elements, such as clickable buttons, menus, text, or headings. These general reference to parts of the page must be transformed into precise selectors for the elements in question. For example, Assertion #4 develops a selector that identifies certain elements intended for screen-reader users, while Assertion #11 identifies dropdown menus on a certain page. Second, accessibility guidelines commonly reference general design principles and concepts, such as visual importance, readable text width, and spacing. These design principles must be rigorously formalized. For example, Assertion #6 models visual importance by text size, while Assertion #8 identifies an appropriate spacing factor between lines of text. Identifying relevant elements and concretizing design concepts can be challenging is some cases, but that difficulty reflects the inherent complexity of visual design. Luckily, Visual Logic provides an expressive language for formalizing accessibility and usability guidelines, and so makes it possible to express even rich and complex visual specifications.

4.3.1 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:

$$\forall b \in \mathcal{B} : \mathsf{page_content}(b) \implies b.\mathsf{height} \geq 14$$

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 essential page content.[6] 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

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

be identified by the "#footer" or ".footer" selectors.

$$\mathsf{page_content}(b) = b.\mathsf{type} = \mathsf{text} \wedge \neg b.\mathsf{whitespace} \wedge \neg(\mathsf{is_descendant}(b, (\#\mathsf{footer}, .\mathsf{footer})))$$

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. Visual Logic allows describing properties that must be true for a range of font size preferences (such as 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 51 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:[7]

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

3. Lines are no more than 80 characters

Users with difficulty seeing sometimes find it hard to keep their place in a long line of text. The 51 evaluation pages are all in English, where 80 characters is a commonly used rule of thumb for line length [W3C08]:

$$\forall b \in \mathcal{B} : \mathsf{page_content}(b) \implies \mathsf{line_width}(\mathsf{line}(b)) \leq 80 \times \mathsf{char_width}(b)$$

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

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

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

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. Content intended solely for such users is often positioned off-screen, where sighted users will not see it, and they ought to 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:

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

We found that a captions for social sharing buttons were a common form of screen-reader-only text. Many websites' social sharing buttons are implemented by setting the buttons' background-image property to those social networks' logos. Since background images are not accessible to blind users, these pages also include text naming the service, but position this text off-screen.

$$\mathsf{for_screenreader}(b) := \mathsf{social_media_caption}(b)$$
$$\mathsf{social_media_caption}(b) := b.\mathsf{type} = \mathsf{text} \land \mathsf{is_descendant}(b, \#S)$$

where S names a social service.[8] Of course, different websites will have other screen-reader-only text; these websites would need a use implementation of the for_screenreader function that selects that text.

[8] In our formalization, Twitter, Facebook, YouTube, Vimeo, Flickr, LinkedIn, Pinterest, Google+, and RSS buttons.

The is_descendant function is simple wrapper around ancestor that checks whether a box descends from an element matching a given selector:

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

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 800–1280 pixels tall to evaluate pages designed for laptop displays, or 320–1920 pixels wide and 320–1280 pixels tall to evaluate pages that support mobile users.

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 change the appearance of these tags, making screen-reader users and sighted users perceive a different hierarchy of importance in page content. The following assertion checks that no such misuse occurs.

$$\begin{aligned} &\forall b_1, b_2 \in \mathcal{B} : \ \mathsf{in_header}(b_1) \wedge \mathsf{in_header}(b_2) \wedge \\ &\qquad \mathsf{header_level}(b_1) < \mathsf{header_level}(b_2) \implies \\ &\qquad\qquad \mathsf{visual_importance}(b_1) > \mathsf{visual_importance}(b_2) \end{aligned}$$

We used text height to establish visual importance on the evaluation pages:

$$\mathsf{visual_importance}(b) := b.\mathsf{height}$$

Selecting headings is done using $\mathsf{is_descendant}$:

$$\mathsf{in_header}(b) := b.\mathsf{type} = \mathsf{text} \wedge \neg b.\mathsf{whitespace} \wedge \mathsf{is_descendant}(b, (\mathsf{h1}, \mathsf{h2}, \mathsf{h3}, \mathsf{h4}, \mathsf{h5}, \mathsf{h6}))$$

Once a heading is selected, its level can be computed from its tag name:[9]

$$\mathsf{header_level}(b) := \textbf{ if } \mathsf{is_descendant}(b, \mathsf{h1}) \textbf{ then } 1 \textbf{ else } \ldots$$

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:

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

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

$$\mathsf{overlaps}(b_1, b_2) = \mathsf{h_adjacent}(b_1, b_2) \wedge \mathsf{v_adjacent}(b_1, b_2)$$

Horizontal adjacency tests three separate cases:

$$\mathsf{h_adjacent}(b_1, b_2) = b_1.\mathsf{right} > b_2.\mathsf{left} > b_1.\mathsf{left} \vee b_2.\mathsf{right} > b_1.\mathsf{left} > b_2.\mathsf{left} \vee b_1.\mathsf{left} = b_2.\mathsf{left}$$

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.

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

[9] Visual logic expands **if** statements and other standard shorthands to the primitive operators of Figure 4.1; see Section 4.2.

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' recommendations of 1.2—because it seems that many of web pages purposely use tight line spacing for visual effect. As an alternative, designers may adapt this assertion to a particular web page by refining page_content to specifically identify running text (as opposed to slogans or call-outs) which are especially important to loosely space.

4.3.2 Page-Specific Assertions

In translating the guidelines of Table 4.1, 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 and web sites of many similar pages, as opposed to evaluation suites of dozens of different pages, these single-page assertions better model the expected use case of Visual Logic.

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 its background in order to make their content more readable for these users.

$$\forall b \in \mathcal{B} : \mathsf{page_content}(b) \implies \mathsf{good_contrast}(\gamma(b.\mathsf{fg}), \gamma(b.\mathsf{bg}^*))$$

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

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:

$$\mathsf{high_contrast}(\mathsf{lum}_1, \mathsf{lum}_2) = \mathsf{lum}_1 \geq 3 \times (\mathsf{lum}_2) \vee \mathsf{lum}_2 \geq 3 \times (\mathsf{lum}_1)$$

This luminance is computed (by a standard formula [W3C08]) for the foreground and background colors of the text.

$$\mathsf{good_contrast}(fg, bg) = fg \neq \mathsf{transparent} \wedge \mathsf{high_contrast}(\mathsf{lum}(fg) + .05, \mathsf{lum}(bg) + .05)$$

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

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

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 assertion, 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. However, web pages frequently place text over part of an image while ensuring that the text is only atop a solid-colored portion of the picture. For example, the web page puppy features a background image with a puppy. Text is then placed atop a blank portion of this image. This Visual Logic assertion requires that the text stay contained to the blank portion.

$$\forall b_1, b_2 \in \mathcal{B} :\ b_1 \in \$(\#\mathsf{background}) \wedge \mathsf{over_bg}(b_2) \implies \neg\mathsf{over_img}(b_1, b_2)$$

The background image can be identified by the background identifier:

$$\mathsf{over_bg}(b) = b_2.\mathsf{type} = \mathsf{text} \wedge \mathsf{is_descendant}(b, \#\mathsf{background})$$

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

$$\mathsf{over_img}(b_1, b_2) = b_1.\mathsf{right}[\mathsf{padding}] - b_2.\mathsf{right} \geq 300$$

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, uses drop-down menus, where the menu is hidden off-screen[10] 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.

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

In this assertion, the first selector matches dropdown menus and the second one ensures that the menu has not been opened (by hovering over it).

12. Columns are vertically aligned

Visual logic can also be used to verify purely aesthetic 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:

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

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 or disappear.

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

---

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

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

$$\mathsf{within}(b_1, b_2) = b_1.\mathsf{left} \geq b_2.\mathsf{left} \wedge b_1.\mathsf{top} \geq b_2.\mathsf{top} \wedge b_1.\mathsf{right} \leq b_2.\mathsf{right} \wedge b_1.\mathsf{bottom} \leq b_2.\mathsf{bottom}$$

14. Main button should be large

Users with motor disabilities have difficulty using small buttons; users on mobile devices have similar problems. Common recommendations [T12] 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.

Chapter 5
FORMALIZING BROWSER LAYOUT

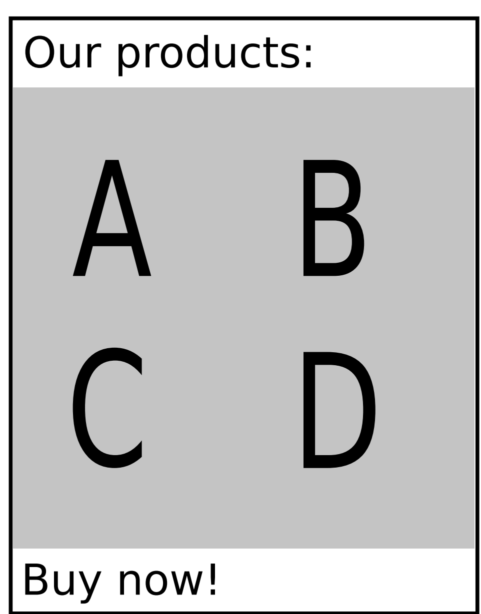
Visual Logic formulas describe properties that a web page layout ought to satisfy at any setting of the browser parameters. To verify that this holds requires understanding the possible layouts that a web page may produce: a formal description of the browser layout algorithm. The Cassius framework provides such a formal description in the form of a mathematical relation between the browser parameters and the generated layout. Cassius describes a fragment of the browser layout algorithm, including core steps like: selector matching and cascading; text, inline, and block layout; flow, positioned, and floating layout; and the calculation of margins, shrink-to-fit widths, and line height. A particular challenge is the formalization of floating layout, where Cassius uses a novel *exclusion zone* data structure to track the areas of the screen where floating boxes are placed.

5.1 The Cassius Framework

Cassius can be thought of as a symbolic browser that exposes an interface similar to that of a standard, imperative browser. Given an HTML document and a CSS stylesheet, and a set of browser parameters, An imperative browser reads an HTML document and CSS stylesheet, and converts a set of browser parameters (the inputs) to a *layout* that displays the document as specified by the stylesheet. This layout is a tree of boxes, annotated with size, position, and color information, Figure 5.1 shows an example document, stylesheet, and layout.

Cassius is based on the same concepts: a tree-structured HTML document, a CSS stylesheet that *styles* the nodes in this tree, and a layout of the tree that respects the given style constraints. Unlike an imperative browser, however, Cassius can perform the layout computation both forwards and backwards: it can not only compute a layout from

```
<html> <body>
  <p>Our products:</p>
  <main>
    <div>A</div> <div>B</div>
    <div>C</div> <div>D</div>
  </main>
  <p>Buy now!</p>
</body> </html>
```




```
main { background:gray; width:100%; float:left }
div { height:200px; width:200px; float:left; font-size:144pt; }
```

Figure 5.1: HTML and CSS for a grid of products. An HTML file (above the line on the left) or *document* defines a tree of elements; the CSS file or *stylesheet* below the line has two rules each of which sets several properties. The resulting layout is shown on the right.

the parameters, but it can also compute the parameters from a desired layout. To enable reversible layout, Cassius operates on *symbolic* parameters and layouts: *sketches* that contain unknown values or *holes* to be filled by the declarative browser. (An imperative browser, in contrast, allows only the layouts to have holes.) Figure 5.2 shows an HTML document and stylesheet (`A` and `B`, in Cassius format), an example input to Cassius `C`, and a symbolic layout `D`. Cassius takes inputs in a simple S-expression syntax, with question marks (`?`) denoting holes. Additionally, holes can have constraints: the `:font-size` parameter is given the value `(between 16 32)`, which denotes a hole constrained to be between 16 and 32 (inclusive). Cassius fills these holes by reducing the layout computation problem to a set of constraints in linear real arithmetic (LRA), solved with Z3 [DMB08]. Solving for the holes in Figure 5.2, for example, is akin to laying out the HTML and CSS from Figure 5.1.

In general, Cassius takes as input a stylesheet, a set of document-layout pairs, a set of symbolic browser parameters, and a set of LRA *assertions* on the holes (expressing, e.g., usability properties). Cassius will then fill the holes in the parameters and layouts so that the resulting concrete stylesheet simultaneously renders each document to its layout, while satisfying the provided assertions. Our declarative browser can thus execute the layout process both forwards and backwards for multiple documents that share the same stylesheet.

A declarative browser provides a convenient platform for building a wide variety of semantic tools for web developers. This work is concerned mainly with verification, but a prototype debugger and synthesizer are discussed in Section 10.3. Such tools are built by reducing a development task—such as repairing a broken stylesheet—to the problem of completing holes in a symbolic stylesheet or layout. Cassius either completes the holes or returns an explanation for why a completion does not exist, given in terms of CSS constraints. The client tool then presents this output to the user.

```
(document A
  ([html]
   ([body]
    ([p] "Our products:")
    ([main]
      ([div] "A")
      ([div] "B")
      ([div] "C")
      ([div] "D"))
    ([p] "Buy now!"))))
```

```
(stylesheet B
  ((tag main)
   [width (% 100)]
   [float left])
  ((tag div)
   [width (px 200)]
   [height (px 200)]
   [float left]))
```

```
(parameters C
  :width (between 800 1200)
  :height (between 600 1024)
  :font-size (between 16 32)
  :scrollbar-width (between 0 15))
```

```
(layout D
  ([ROOT :w 400 :h 600]
   ([BLOCK :w ? :h ? :x ? :y ?]
    ([BLOCK :w ? :h ? :x ? :y ?]
     ([BLOCK :w ? :h ? :x ? :y ?]
      ([LINE :w ? :h ? :x ? :y ?]
       ([TEXT :w 112 :h 19 :x 0 :y 16])))
     ([BLOCK :w 400 :h 400 :x ? :y ?]
      ([BLOCK :w 200 :h 200 :x 0 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 200 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 0 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 200 :y ?] ...))
     ([BLOCK :w ? :h ? :x ? :y ?]
      ([LINE :w ? :h ? :x ? :y ?]
       ([TEXT :w 76 :h 10 :x 0 :y 451]))))))))
```

Figure 5.2: Figure 5.1 translated to Cassius input, containing an HTML file, a CSS file, and a layout (line and text boxes for products are elided for space). All parts of the CSS file or of the layout, except the width and height of text boxes (see Section 5.2.1), can be replaced by holes (?), and Cassius will solve for the values of these holes automatically.

5.2 *CSS Semantics*

A key feature of Cassius is a declarative formalization of a substantial fragment of the CSS semantics. The complete formalization is publicly available in machine-readable form.[1] This section presents the core concepts of web page layout, as formalized in Cassius, highlighting design decisions that enable efficient encoding of the layout process in the theory of (quantifier-free) linear real arithmetic.

Cassius's formalization of the browser layout algorithm is significantly more detailed and conformant than that of previous work, including not just line height, margin collapsing, and floating layout, discussed earlier, but also positioned layout (which allows placing a box at a fixed pixel position on the screen), clearance (which allows moving a box vertically in response to floats), scroll bars, and the interaction of elements with different layout modes.

Cassius derives its specification of CSS layout from the W3C CSS 2.1 standard [W3C07]. The standard describes CSS via an abstract algorithm (Figure 2.1). Given an HTML document and a CSS stylesheet, the algorithm computes a *layout* (2.1d), which is a set of boxes with known position and size. This computation takes as input the document, represented by a tree of elements (2.1a); constructs a tree of *layout boxes* from the element tree and stylesheet rules (2.1b); and arranges the resulting boxes into *flow trees* (2.1c), from which sizes and positions for each box (i.e., the layout) can be computed. Cassius is a specification of this abstract layout algorithm, describing the values computed for each box and the rules by which these values are computed.

The Cassius specification is declarative: it specifies a relation on the HTML elements $\mathcal{E}$, CSS rules $\mathcal{R}$, the browser parameters $\mathcal{M}$, and the boxes $\mathcal{B}$ that form the final layout. As demonstrated in Section 8.3, such a declarative formalization provides a versatile platform for building semantic tools: it can be used for verification (by specifying $\mathcal{E}$ and $\mathcal{R}$ and solving for $\mathcal{B}$), synthesis (by specifying $\mathcal{B}$ and $\mathcal{E}$, and sketching $\mathcal{R}$), or debugging (by specifying $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{B}$, and computing unsatisfiable cores). In the rest of this section, we describe

---

[1] At `https://github.com/uwplse/cassius`

the fragment of CSS implemented by Cassius; the representation of $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{B}$; and our formalization of the CSS layout algorithm.

5.2.1 Supported Fragment of CSS

Cassius specifies layout for a fragment of CSS level 2.1. The specification covers the cascading rules and the box model, including: box generation; block, line, and inline boxes; floating, positioned and in-flow layout modes; along with the specific layout features diagrammed in Figure 5.3. This forms a substantial fragment of CSS, and suffices to lay out fragments of real-world websites. With the fragment of CSS currently specified, additional CSS features could be added in a straightforward manner.

The Cassius formalization focuses on the core layout algorithm. It also omits the modeling of layout features that rely on dictionaries and other linguistic information (such as font metrics, line breaking, and hyphenation), which are difficult to formalize but easy for a client tool to provide. Finally, the CSS standard describes a rich universe of CSS selectors; Cassius models only a subset. Our implementation simply ignores unknown CSS properties—they can appear in an input to the Cassius declarative browser, but their semantics will not be considered by the underlying SMT solver.

5.2.2 Representing Elements, Rules, and Boxes

Cassius specifies CSS layout as a relation between a tree of elements $\mathcal{E}$, CSS rules $\mathcal{R}$, the parameters $\mathcal{M}$, and the layout $\mathcal{B}$. The input to Cassius is a representation of these four entities. The elements $\mathcal{E}$ are given by an abstract syntax tree (Figure 5.2a), where each node is labeled with an HTML *tag* and an optional *identifier*, or is a string of text. The CSS rules $\mathcal{R}$ are given by a list of rule blocks (Figure 5.2b), each of which contains a *selector* and a collection of property-value pairs. The selector identifies the elements in $\mathcal{E}$ that are styled by a given rule. The properties can be given as *holes*—undefined values that Cassius will solve for. Parameters $\mathcal{M}$ are given as ranges of real-number values, one per parameters; ranges with equal endpoints correspond to concrete values, while ranges with different endpoints

correspond to holes with inequality constraints on those holes' values. Finally, the layout $\mathcal{B}$ is represented by a set of boxes, arranged in a tree (Figure 5.2c). Each box is annotated with the box type (root, block, inline, line, or text); the element $e \in \mathcal{E}$ from which it was generated;[2] its position, width, and height; and the width of the border on each side. Any of these fields can be holes except width and height on text boxes.[3]

5.2.3 Layout phases

Numerous browser subsystems interact to render HTML and CSS (Figure 5.3):

- *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.
- *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.

---

[2]Some boxes are not generated by an element, such as line and text boxes or "anonymous" block box. These boxes are annotated as anonymous.

[3]Since Cassius does not model font metrics, the width and height of text boxes must be provided by tools that build upon Cassius, for example by rendering the text and measuring its size.

- *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 6.3.1.

- *Margins* sometimes *collapse*, allowing adjacent vertical margins of boxes to overlap, following rules discussed in Section 6.3.2.

- *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 5.3 describes this subsystem.

Cassius implements each of these phases. To ensure that its implementation is correct, it is extensively validated against Mozilla Firefox (Section 9.1).

5.2.4 Formalization of the Layout Algorithm

The Cassius specification of CSS layout follows the high-level structure of the abstract layout algorithm described in the CSS standard.

CSS Rules A rule $r \in \mathcal{R}$ is a map from a subset of the CSS properties $\mathcal{P}$ to a value for each property. We write $p \in r$ to denote that the rule r specifies a value for the property p, and we write $r[p]$ for that value. The value $r[p]$ is either the distinguished constant inherit or it is drawn from a property-specific type. Each rule has an associated selector, $r.\mathsf{selector}$,

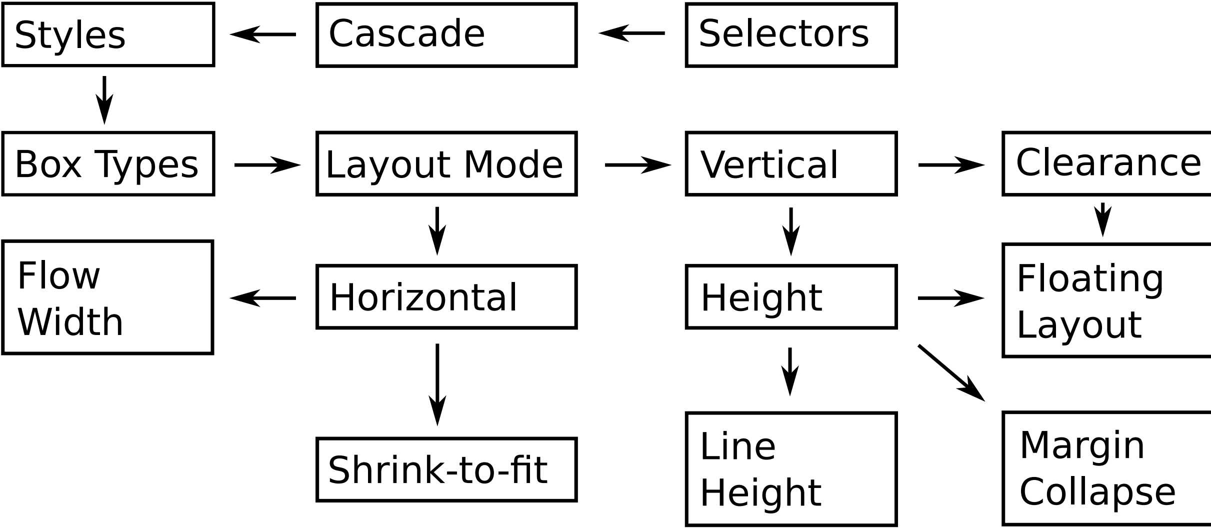


Figure 5.3: The major subsystems of the Cassius formalization of web page layout. Each component is described briefly in Section 5.2.3. Many details, such as Cassius's handling of color, media queries, and width and height bounds, are hidden in this picture, which shows only the highest-level components. Components outside the shaded area are newly formalized in this work.

which can be the universal selector $*$, an HTML tag, an identifier, a class name, a child or descendant operator, or any of a number of other options. Selectors define which elements a rule applies to. Rules also have a media-query guard $r.\mathsf{media}$, which uses the browser parameters like width and height to determine whether the rule is used. Finally, rules are associated with metadata, such as their position in the stylesheet, that influences cascading, as described below.

Elements Every element has a tag and optional attributes, such as its identifier and the classes it is a part of; images also have attributes specifying the image dimensions. A tag specifies the type of an element, while the attributes are used for resolving selectors and during layout: the wildcard selector applies to all elements, and the tag name and identifier selectors apply to those elements that have the given tag name or identifier, and so on for other types of selectors. Elements are arranged in an ordered tree, giving the usual meaning to the notions of the next, previous, first, and last siblings of an element. Each element also has a *computed style*, which maps every CSS property to a value, *not* including the distinguished inherit value.[4] We write $e[p]$ for the computed value of the property p for the element e. This value is derived from the rules $\mathcal{R}$ through a process called *cascading*.

Cascading The cascading process determines the computed value $e[p]$ of every element $e \in \mathcal{E}$ and property $p \in \mathcal{P}$. If no rule in $\mathcal{R}$ both applies to e and specifies a value for p, then $e[p]$ is set to a property-specific default value. Otherwise, $e[p] = r[p]$, where r is the highest scoring rule that sets the property p and whose selector matches e, ranked according to the scoring function from Section 6.4.3 of the CSS standard. This function uses rule metadata to break ties between rules with equally specific selectors. The inherit value for CSS properties is also resolved during cascading.

[4]Note that unlike rules, computed styles are per-element, so they reflect not only the resolution of inheritance but also the resolution of conflicts between multiple rules.

Boxes The CSS standard defines three types of boxes—block, line, and inline. For modeling convenience, Cassius defines four additional types of boxes: root boxes, which represent browser windows; text boxes, which abstract text rendering; and *opaque boxes*, which abstract the layout of elements (e.g., tables) that are outside of our fragment of the CSS semantics. The root and text boxes are implicit in the CSS standard.

Like elements, boxes are arranged in an ordered tree, giving each box a parent, children, and siblings. Every box also has a width, height, $(x, y) \in \mathbb{R}^2$ position, and an element from which it was generated. These values are computed based on the *margin*, *padding*, and *border* width in each direction, as specified by the *CSS box model.* Cassius models all parts of a box explicitly, as well as the rules for computing box layout, which differ for block boxes, inline and text boxes, and line boxes.

Block Boxes Block boxes can be either *in-flow* or *floating*, and they are generated from block-level elements, such as paragraphs (`<p>`). In-flow boxes take up all the available horizontal space (in their parent container) and are laid out one after another vertically. Floating boxes are placed to the left or right of other boxes, with text flowing around them.

The CSS standard prescribes a complex set of rules for computing the positions and sizes of block boxes. The computation involves partitioning the box tree into a forest of *flow trees*, computing *used values* of box properties, and collapsing margins of certain boxes. Imperative browsers must carefully sequence this computation. For example, the width of boxes is determined in a top-down pass, while their height is determined in a bottom-up pass, since the height of an element might depend on the size and position of its children.

Because the Cassius semantics is declarative, it dispenses with the need for sequenced computation of box properties. Instead, it simply constrains the final value of each property with a formula. For example, Cassius specifies the x position of an in-flow block box to be the sum of the parent's x position, left padding and border, plus the box's left margin:

$$\forall b \in \mathcal{B}, \mathsf{block\text{-}box}(b) \land \mathsf{in\text{-}flow}(b) \implies b.\mathsf{x} = b.\mathsf{parent.left\text{-}content\text{-}edge} + b.\mathsf{margin\text{-}left}$$

Our machine-readable specification of block layout distills 36 pages of the CSS standard into just 790 lines of code.

Line Boxes Line boxes are generated when a block box contains inline or text boxes. Each line box represents a line of text, and all of its siblings are also line boxes. Usually, a line box spans from the left to right edge of its parent's content area. However, line boxes shrink to avoid floating boxes. For example, the following constraints forces line boxes to avoid block boxes that float to the right:

$$\begin{aligned}\forall b \in \mathcal{B}, f \in \mathcal{B}, \mathsf{line\text{-}box}(b) \wedge f.\mathsf{element}[\mathsf{float}] = \mathsf{right} \wedge \\ \mathsf{is\text{-}preceding\text{-}floating\text{-}box}(f, b) \implies \\ \textbf{if } b.\mathsf{top\text{-}margin\text{-}edge} < f.\mathsf{bottom\text{-}margin\text{-}edge} \\ \textbf{then } b.\mathsf{right\text{-}margin\text{-}edge} = f.\mathsf{left\text{-}margin\text{-}edge} \\ \textbf{else } b.\mathsf{right\text{-}margin\text{-}edge} = p.\mathsf{right\text{-}content\text{-}edge}\end{aligned} \tag{5.1}$$

Line boxes also align their children to the left, right, or center, depending on their parent's text-align property.

Browsers generate line boxes with a *line breaking* algorithm, which breaks text into lines to achieve a visually pleasing result. Different browsers use different line breaking algorithms, and the CSS 2.1 standard does not specify any constraints on this algorithm. As a result, Cassius does not constrain line breaking in any way, instead relying on the input to contain pre-broken lines.[5] This modeling choice does not affect either synthesis or debugging, since the full layout (including the desired line breaks) is available in these tasks. It may, however, cause a verification tool to produce false positives—layouts that violate a desired property under our declarative semantics but not on any imperative browsers, which constrain line breaking. We have found such false positives to be rare in practice (see Section 9.2).

[5]Since line breaking depends on the fonts used on the page, the language the page is written in, and dictionaries of language-specific layout rules, modeling line breaking in Cassius would be difficult and would likely cause severe performance degradation.

Inline and Text Boxes Inline and text boxes are always descendants of line boxes. They are laid out left to right, each lying to the right of the previous box. Cassius specifies this horizontal layout as follows:

$$\forall b \in \mathcal{B}, \mathsf{text\text{-}box}(b) \wedge \mathsf{is\text{-}box}(b.\mathsf{previous}) \implies b.\mathsf{left} = b.\mathsf{previous}.\mathsf{right}$$

Cassius does not model the height and width computation for text boxes, which requires access to font metrics. These values must be provided by client tools. CSS requires left (and right) padding, margins, and borders to only apply to the first (and, respectively, last) boxes generated by an inline element when that inline element is split over multiple lines. Layout of inline boxes thus requires checking whether the box is the first or last box generated by its element, and applying margins, borders, and padding accordingly.

Opaque Boxes Many websites use features of CSS that are outside the subset that Cassius supports. To enable client tools to reason about these features, Cassius provides an escape hatch from its CSS semantics: an element can generate an opaque box, whose position is not related to any CSS properties or to any other boxes. Whether an opaque box floats, or has any other properties relevant for layout, is also not constrained. This lenient encoding of constraints on opaque boxes ensures that we allow any likely behavior of unknown CSS properties. Client tools can use assertions to constrain the behavior of these boxes as needed.

5.3 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" [MDN17]. Floating layout is commonly used to implement sidebars, figures, and menus. Floating layout is particularly challenging and Cassius's formalization is, as a result, particularly novel. To formalize floating layout, Cassius reduces the set of all preceding floating boxes to a data structure called an *exclusion zone*.

5.3.1 *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.

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. These rules are listed in section 9.5.1 of the standard [W3C11]:

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.

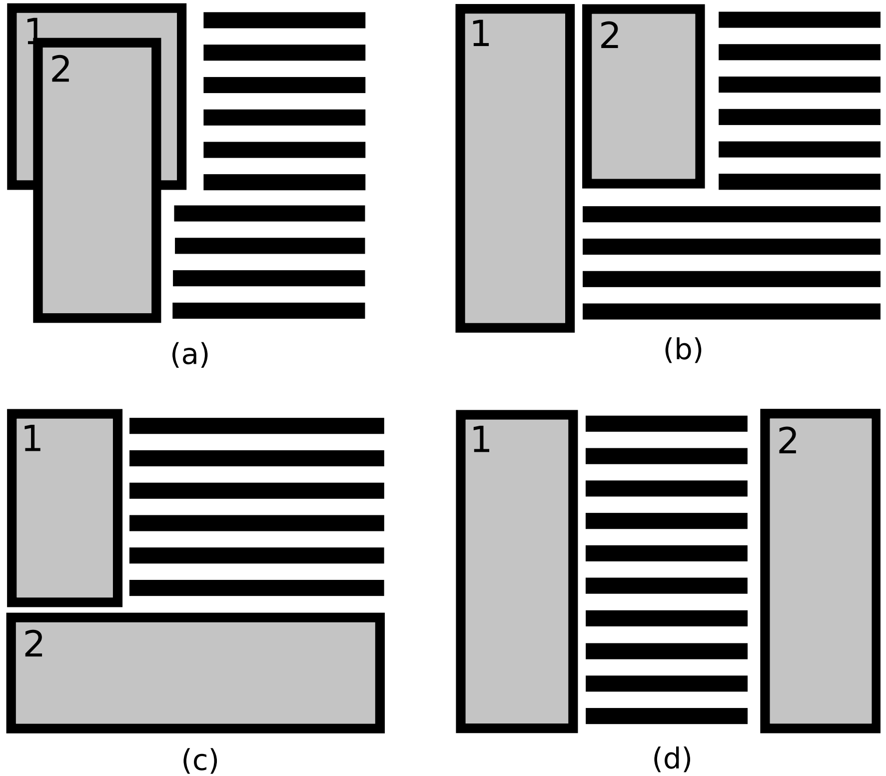
9. A left-floating box must be put as far to the left as possible, a right-floating box as far to the right as possible. A higher position is preferred over one that is further to the left/right.

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.

5.3.2 Restrictions on Floating Boxes

Because of these problems, Cassius initially only allowed restricted uses of floating layout. (The next subsection introduces *exclusion zones*, which allowed Cassius to lift these restrictions.)

While it is possible for a direct translation of the 9 standards rules to fully specify the behavior of floating boxes, such a specification would not be amenable to efficient automated reasoning. Because a floating box may affect the layout of every other box, and every block element can generate floating boxes, expressing these interactions in a quantifier-free logic would be prohibitively expensive. In particular, every float constraint in our semantics (such as Rule 5.1 in Section 5.2.4) would lead to $O(|\mathcal{B}|^2)$ constraints in a quantifier-free logic.




```
<div>1</div><div>2</div>Text text text text text text text...
```

Figure 5.4: Interactions between floating boxes disallowed in Cassius. Each diagram is a possible layout of a document containing two floating boxes followed by text. In (a), the two floating boxes overlap, and the text must wrap around both. In (b), two floating boxes stack horizontally where the later box is smaller than the previous box. In (c), the second box is so wide it must wrap to the next line, but text continues to fit beside the first box. In (d), text flows around a left-floating and right-floating box. By disallowing these four types of interactions, text layout only relates a text box to its previous floating box. Note that any layout is achievable despite these restrictions, as long as the HTML is modified to rearrange the order of elements or add an extra block element around the text and the second `div`.

To enable efficient automated reasoning, the first Cassius semantics imposed four restrictions on floating boxes, illustrated in Figure 5.4. The restrictions ensure that the layout of every box depends only on the layout of the floating box closest to it in an in-order traversal of the box tree, rather than on the layout of all floats. We did not believe these restrictions to be onerous, since the forbidden layouts can be often be achieved by adding extra elements to the document. However, later work on professional web page templates Section 9.2 required more comprehensive support for floating layout and thus a lifting of these restrictions.

5.3.3 Exclusion Zones

Exclusion zones instead allow full support for semantics of floating layout. 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. 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 $r \in R$,

$$p.\mathsf{y} < y \vee (p.\mathsf{x} < \ell.\mathsf{x} \wedge p.\mathsf{y} < \ell.\mathsf{y}) \vee (p.\mathsf{x} > r.\mathsf{x} \wedge p.\mathsf{y} < r.\mathsf{y})$$

Figure 5.5 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

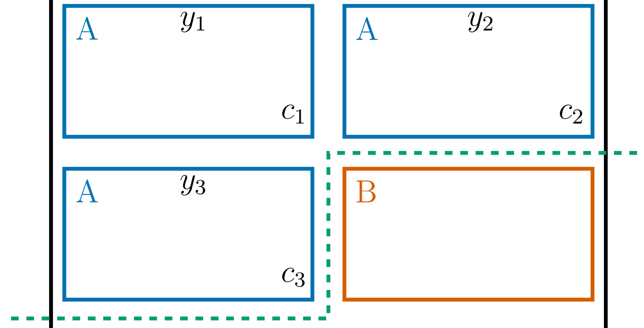


Figure 5.5: A left-floating box is laid out in the topmost, leftmost position outside an *exclusion zone*. Here, the blue boxes labeled "A" are left-floating children of their parent, whose left and right edges are shown. The resulting exclusion zone is all points above (and left of) the green dotted 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, $(\{y_3\}, \{c_3\}, \emptyset)$.

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.

$$0 \leq w \leq p.\mathsf{width}[\mathsf{content}] \implies w \leq r - \ell$$

$$r = \min\{x_r \mid (x_r, y_r) \in R \land y < y_r\}$$

$$\ell = \max\left(\{x_l \mid (x_l, y_l) \in L \land y < y_l\} \cup \{p.\mathsf{left}[\mathsf{content}]\}\right)$$

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, Cassius 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. Simi-

larly, the x position must (for a left float) be a member of $\{x_l \mid (x_l, y_l) \in L\} \cup \{p.\mathsf{left}[\mathsf{content}]\}$.

Chapter 6
AUTOMATING VERIFICATION

Using CSS to specify an attractive and usable web page layout can be surprisingly tricky. Even expert-written CSS (e.g., Bootstrap [OT15]) can suffer from usability bugs, such as overlapping text or control elements (buttons), figures separated from their captions, or elements rendered off-screen on smaller devices. These bugs may only manifest at a particular screen size, making them difficult to discover. In practice, developers spot-check for stylesheet bugs manually, by using an imperative browser to render their web page at a few manually-selected screen sizes, text sizes, etc. Such manual testing is both tedious and unlikely to achieve good coverage.

6.1 How VizAssert uses Cassius

Using a declarative browser, we can instead *verify* that web page layout desired properties. Such a verifier, VizAssert, is built atop the Cassius framework.

VizAssert takes as input a web page, a set of browser parameters and ranges for their values, and a property P, a verifier proves that P is satisfied when laying out the page under all possible values of browser parameters from the given ranges. Figure 6.1 shows an example stylesheet (6.1a) and property P (6.1b) consumed by VizAssert. VizAssert passes the web page (with a fully symbolic layout) and holes for the browser parameters to Cassius, along with the assertion $\neg P$ on the holes in the browser parameters. Cassius then searches the resulting declarative layout problem for a completion of the holes that violates P. If a completion exists, VizAssert converts it to a concrete counterexample layout (Figure 6.1d) and presents the counterexample to the developer. Otherwise, all possible layouts of the page are guaranteed to satisfy P.

```
(stylesheet B
  ((tag main)
   [width (% 100)])
  ((tag div)
   [width (px 200)]
   [height (px 200)]
   [float left]))
```

(a)

$$\forall a \in \mathcal{B}, b \in \mathcal{B},\quad a \in \$(\mathsf{main}) \land b.\mathsf{parent} = a \implies \mathsf{contains}(a, b)$$

(b)

```
(layout C
  ([VIEW
    :w (between 200 1920)]
   ([BLOCK ...]
    ([BLOCK ...]
     ([BLOCK :w ? ...] ...)
     ([BLOCK ...] ...)
     ([BLOCK ...] ...)))))
```

(c)

```
(layout counterexample
  ([VIEW :w 400]
   ([BLOCK ...]
    ([BLOCK ...]
     ([BLOCK ...] ...)
     ([BLOCK :w 400 :h 0
       :x 0 :y 16]
      ...)
     ([BLOCK ...] ...)))))
```

(d)

Figure 6.1: A prototype CSS verification tool built on top of Cassius. The same HTML document as in Figure 5.1 is used, but with an incorrect stylesheet (a) that fails to guarantee that the container expands to contain all products (b). Given a choice of the browser window width (c), the verifier finds a counterexample (d) where the property does not hold.

6.2 SMT Encoding

Cassius specifies the semantics of CSS layout declaratively, as a set of formulas in the machine-readable SMT-LIB2 [BFT15] format. This specification closely parallels the description given in Section 5.2: quantified formulas define layout constraints on all boxes or all elements, with layout rules for different types of boxes encapsulated in functions. Such a high-level encoding has the advantage of being easy to audit for correspondence to the CSS standard. But it does not consistute a practical implementation of a declarative browser.

To provide a practical framework for building solver-aided layout tools, Cassius reduces the high-level semantics of a given declarative layout problem to the theory of quantifier-free linear real arithmetic (QF-LRA). Cassius's inputs are used to instantiate (or, ground) the quantifiers in the high-level semantics, yielding a set of quantifier-free LRA formulas. The resulting formulas are then simplified, fed to an off-the-shelf SMT solver (Z3 [DMB08]), and the solver's output is used to either fill the holes in the input or explain why a solution does not exist. We describe these steps in more detail below.

6.2.1 Grounding

In principle, it is easy to produce a quantifier-free encoding of the Cassius semantics for a given web page D, and a layout B. We simply expand each universally quantified layout rule into a conjuction of grounded (quantifier-free) constraints. For example, a rule of the form $\forall b \in \mathcal{B}, e \in \mathcal{E}, F(b, e)$ becomes a ground formula of the form $\bigwedge_{x \in B, y \in D} F(x, y)$, where x and y range over all boxes in B and elements in D, respectively. In practice, however, naive grounding produces a large formula that overwhelms the solver.

For example, consider the problem of describing the size of an absolute-positioned element. The size of such an element is related to the size of its *containing block*, which might be its parent, its grandparent, or some even more ancestral relation. The high-level semantics expresses this computation by specifying layout rules in terms of a containing block relation, is-containing-block. The following rule computes the width (w) of a block box from width of

its containing block:

$$\forall b_1, b_2 \in \mathcal{B}, \textsf{is-containing-block}(b_2, b_1) \wedge \textsf{absolutely-positioned}(b_1) \wedge \\ \textsf{is-percentage}(b_1[\textsf{width}]) \wedge b_1[\textsf{left}] = b_1[\textsf{right}] = \textsf{auto} \implies \\ b_1.\textsf{w} = b_2.\textsf{w} \times b_1[\textsf{width}].\textsf{percentage} \quad (6.1)$$

Instead of reducing such rules into equivalent ground formulas of size $O(|\mathcal{B}|^2)$, Cassius reduces them into equisatisfiable ground formulas of size $O(|\mathcal{B}|)$, by introducing a small set of *auxiliary uninterpreted functions*.

The key idea is to use the auxiliary functions to rewrite each high-level layout rule into a formula with at most one universal quantifier (over boxes). For example, containing-block is an auxiliary function that maps each box to its float predecessor. Using containing-block, Cassius rewrites our sample rule (6.1) into the following formula:

$$\forall b_1 \in \mathcal{B}, \ \textbf{let } b_2 = \textsf{containing-block}(b_1) \textbf{ in } \textsf{absolutely-positioned}(b_1) \wedge \\ \textsf{is-percentage}(b_1[\textsf{width}]) \wedge b_1[\textsf{left}] = b_1[\textsf{right}] = \textsf{auto} \implies \\ b_1.\textsf{w} = b_2.\textsf{w} \times b_1[\textsf{width}].\textsf{percentage} \quad (6.2)$$

The resulting formula is then simply ground into a conjunction of size linear in $|\mathcal{B}|$.

To ensure that this transformation preserves equisatisfiability, Cassius constrains the interpretation of each auxiliary function with suitable axioms. Thanks to a careful selection of auxiliary functions, these axioms can also be expressed using at most one universal quantifier. For example, containing-block is constrained for all b by:

$$\begin{array}{lll} \forall b \in \mathcal{B}, & \textbf{if} & \textsf{is-containing-block}(b.\textsf{parent}, b) \\ & \textbf{then} & \textsf{containing-block}(b) = b.\textsf{parent} \\ & \textbf{else} & \textsf{containing-block}(b) = \textsf{containing-block}(b.\textsf{parent}) \end{array}$$

Thanks to the choice of auxiliary functions, the ground encoding as a whole—including both the transformed rules and the auxiliary axioms—is linear in $|\mathcal{B}|$.

The effectiveness of our grounding approach hinges on finding a small set of auxiliary functions that are both efficiently axiomatizable and sufficient to eliminate nested quantification (over boxes) from the high-level semantics. Intuitively, this means choosing a subset of CSS that can be laid out with a *non-deterministic, incremental* version of the abstract layout algorithm (Figure 2.1) described in the CSS standard. The auxiliary functions introduced by Cassius encode the auxiliary information that the incremental algorithm would need to track in order to compute the layout in one pass, given a correct (angelically chosen) order in which to traverse the tree of boxes. Our grounding approach can thus be understood as encoding an arbitrary execution of this incremental algorithm on a specific stylesheet, documents, and layouts.

6.2.2 Constraint Simplification and Solving

In the final stages of declarative layout, Cassius simplifies the ground encoding of the semantics using a custom formula optimizer, and passes the resulting constraints to Z3. The optimizer avoids simplifications that interfere with unsatisfiable core extraction, and embeds metadata into the encoding (using Z3's named constraints) that relate the optimized and ground semantics. This information is used to lift the solver's output into a solution to the declarative layout problem received by Cassius.

One important change made by the constraint optimizer is in the handling of multiplication. Since QF-LRA is restricted to linear arithmetic, multiplications between variables, such as those needed for percentage computations, are not allowed. Luckily, since in Cassius the stylesheet is generally known, the set of possible percentage values, and thus the set of possible multiplicands, is statically known. This allows replacing multiplication with a custom function, like so:

$$a \times b \leadsto \begin{cases} 1 \times b & \textbf{if}\, a = 1 \\ 0.5 \times b & \textbf{if}\, a = 0.5 \\ \vdots & \end{cases}$$

where there is one case for every value of the multiplicand a. This large if-else block contains only constant multiplications, and is of finite size since all possible values of a are known; thus, the multiplication is placed back within LRA.

6.3 Finitization Reductions

Another key challenge to efficiently reasoning about the Cassius formalization of browser rendering is handling arbitrary-size sets. Solver cannot reason efficiently about arbitrary-size sets, since there are infinitely many sets of arbitrary size. Reasoning about infinitely many possible arbitrary-size sets would require induction and therefore invariant synthesis, which is beyond the abilities of existing solvers. Our methodology avoids this problem by reducing reasoning about an arbitrary-size set to equivalent reasoning about a finite data structure. Cassius uses these *finitization reductions* to formalize many layout concepts. This section describes the formalization of line height computation and margin collapsing using finitization reductions. The exclusion zones used for floating layout can also be seen as an application of this approach, and Cassius also uses finitization reductions for handling percentage dependency loops, computing containing boxes, and resolving inheritance.

6.3.1 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.[1]

[1] Despite its name, the line-height property does not directly set the height of the line box; instead, it sets the leading (the gap above and below the text between individual lines of text). 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.

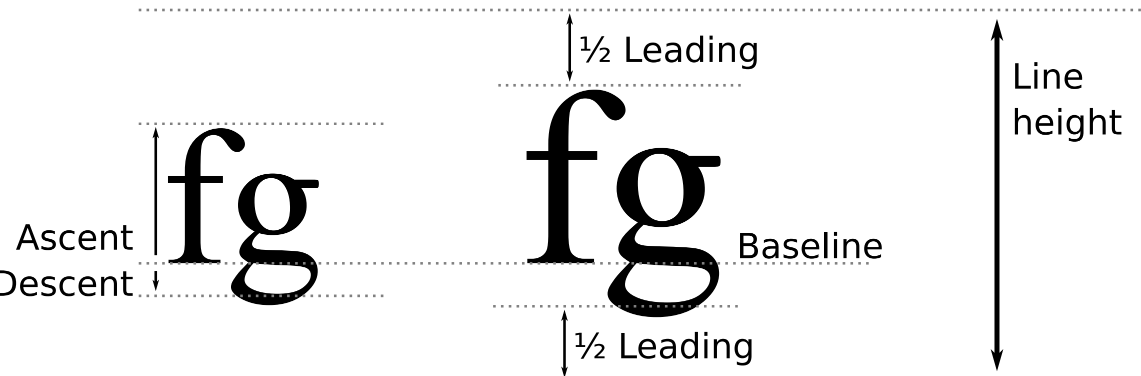


Figure 6.2: Line height is computed by aligning all text in the line to a common baseline and finding the maximum ascents above and descents below the baseline, including blank space above and below the text called leading.

Standards text

The CSS 2.1 specification defines the rules for line height in section 10.8 [W3C11] using the following algorithm (quoted from the standard, with references removed):

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'.

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).

3. The line box height is the distance between the uppermost box top and the lowermost box bottom. (This includes the strut.)

A key concept in properly placing line boxes is to compute the "leading", or gap between successive lines, which the standard defines so:

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 $\mathsf{A} + \mathsf{D} < \mathsf{lh}$ the leading is negative, which is allowed.

Cassius Formalization

The CSS standard requires the browser layout 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 6.2). The foregoing description of line height is not amenable to mechanical reasoning. First, it introduces nested quantifiers by reasoning about the space of possible line heights. Second, it describes the set of all objects in a line, a set too complex for efficient mechanical reasoning. Cassius 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). Cassius computes the highest an object rises above the baseline with a running maximum (and similarly for the deepest an object falls below the baseline):

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

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. Cassius thus reduces line height reasoning to just two values: above_baseline and the symmetric below_baseline.

6.3.2 Margin Collapsing

In CSS, every box has a *margin*: an area outside the box that other boxes may not intrude into.[2] For most boxes, vertical margins are allowed to overlap if adjacent; Figure 6.3 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.

Standards text

The CSS 2.1 specification defines the rules for margin collapsing in section 8.3.1 [W3C11]:

> 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 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

[2] This description elides many details; in some cases, other boxes may intrude into a box margin.

  - 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.

These rules may look somewhat haphazard and arbitrary. This is because they embed a deep understanding of the sorts of designs that web designers look to achieve, designs that themselves draw from a historical tradition of European typesetting that had little reason to conform to simple and mathematically elegant rules. In any case, arbitrary or not, Cassius must model this semantics faithfully.

Cassius formalization

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. Cassius thus uses a finitization reduction that replaces the set of collapsed margins by a finite description. Cassius tracks, instead of the set of collapsed margins, a running maximum of margins seen

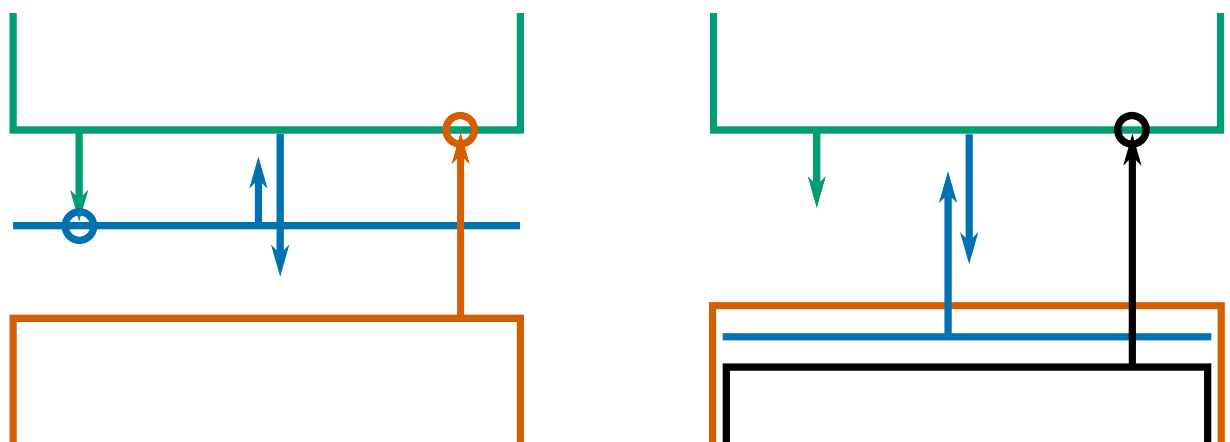


Figure 6.3: Adjacent vertical margins collapse, and this figure visualizes two possibilities. Each possibility shows several boxes and their margins. Zero-height boxes are represented by horizontal lines. Each arrow represents a margin (all positive) and has the same color as the box that generates it. 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.

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 layout algorithm considers positive and negative margins separately, so Cassius 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.

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 $\mathsf{mt}^{\uparrow}_+$ and $\mathsf{mt}^{\uparrow}_-$ values: $\mathsf{mt}^{\uparrow}_+$ is equal to mt_+ for boxes with non-zero height, and is equal to the next sibling's $\mathsf{mt}^{\uparrow}_+$ 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 $\mathsf{mb}^?$ 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_-, $\mathsf{mt}^{\uparrow}_+$, $\mathsf{mt}^{\uparrow}_-$) and the boolean $\mathsf{mb}^?$.

6.4 Reasoning about Floating Layout

Exclusion zones, which model floating layout in Cassius, have minimal *canonical forms* whose compact size makes them an attractive choice for representing floating layout. Cassius

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 Cassius adds to a canonical form, it tests whether the bound is reached; if so, Cassius 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.

6.4.1 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:

$$(x, y) \in e \Leftrightarrow (x, y) \in e' \Leftrightarrow y < 0 \vee x < 1 \wedge y < 1$$

A compact canonical form can be computed:

Theorem 6.1. *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

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

L^* is the maximal subset of L such that

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

R^* is analogous. Checking the three properties given for canonical forms is mechanical. $\square$

Example 6.1. *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 $(\{0\}, \{(1,1), (2,1), \dots (\ell, 1)\}, \emptyset)$. 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 that we have tested with VizAssert (see Section 9.2). 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 as there are floating boxes on the page, this process will always terminate.

6.4.2 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 together and in order of increasing y. To avoid encoding a sort and merge operation in an SMT formula, L and R are stored together: L and R are stored as a size-k list of triples (y, ℓ, r), where each triple represents $(\ell, y) \in L$ and $(r, y) \in R$. Note that there may as many as $|L| + |R|$ or as few as $\min\{|L|, |R|\}$ triples, depending on how

many y values are shared between L and 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. Particularly important is the sorted, paired order in which L and R are stored, which avoids encoding inefficient operations such as sorting and merging that would otherwise be used to iterate over those sets in a sorted order. 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.

6.4.3 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 have performed a paper proof that this automated proof always succeeds.

Chapter 7

MODULAR REASONING

While VizAssert can automatically verify web page layout properties, it does so by considering the page as a whole. Troika likewise verifies web page layout properties, but enables a more modular approach. The developer partitions a web page into components and writes *component specifications* which summarize the possible layouts of each component. Each component specification can be verified independently, using a mix of different verification tools. Troika then proves the layout properties from those specifications. This modular approach allows mixing different verification tools, incremental re-verification, and proof reuse across different pages.

7.1 Modular Layout Proofs

A modular layout proof relies on a decomposition of a web page layout into *components*, whose layout can be separately specified and independently verified. However, unlike prior domains where modularity has been used to scale verification, web pages have no natural computational units such as function or module boundaries. Troika uses a novel definition of web page components, which combines subtrees of the web page HTML with symbolic computed styles summarizing its CSS. This section defines web page components and modular layout proofs, in which *component specifications* constrain the layout of components, and provides an algorithm for checking that a modular layout proof is well-formed. Section 7.2 describes how component specifications are checked.

7.1.1 Finding Modularity in Web Pages

Troika uses a partitioning of the HTML tree into subtrees with holes[1] as its basis for dividing a page into components. Components do not overlap, so every element is in exactly one component. Unlike the HTML tree, a page's CSS rules cannot be partitioned and split between its various components: each CSS rule can apply to multiple elements, including elements in multiple components. Moreover, determining which CSS rules apply to which element requires the full page HTML: it cannot be determined from a component alone. So, a component also includes a symbolic computed style to summarize the CSS that applies to each element.

The *computed style* [W3C11] for an element gives a value for each of its CSS properties. Browser implementations derive computed styles from the page's CSS, and then use that computed style (rather than the original CSS) when laying out the element. Troika does the same, and each component includes its elements' computed styles. Since access to browser parameters is necessary for deriving computed styles, Troika includes a *symbolic* computed style engine. That engine produces, for each element, a list of possible computed values with symbolic *guards* for each, where each guard is a set of linear inequalities over the browser parameters that define when that computed value is to be used. Those guards are used by the solver to define the element's computed style in terms of browser parameters.

Definition 7.1. *A* ***component*** *of a web page is a subtree (with holes) of the page's HTML and symbolic computed style for each element in that subtree. A symbolic computed style is a function from browser parameters to a computed style.*

Definition 7.2. *A* ***decomposition*** *of a web page into components is a set of components of that page that form a partitioning of the page.*

In Troika, a proof author specifies a decomposition by providing a CSS selector for the root element of each component. Each element is placed in the component of its closest

[1] That is, a subtree of the HTML tree, optionally with some subtrees replaced by holes.

selected ancestor. This scheme guarantees that components always form a partitioning of the HTML tree. Assigning a *component specification* P_c to each component yields a modular layout proof:

Definition 7.3. *A **modular layout proof** of a property Q for a web page p consists of a decomposition $\mathcal{C}$ of p and a* component specification *P_c for each component $c \in \mathcal{C}$. A modular layout proof of Q is* well-formed *when $(\bigwedge_c P_c) \implies Q$ as a matter of pure logic.*

Both the web page property Q and the component specifications P_c are given in Visual Logic (Section 4.1). To preserve modularity, a component specification quantifies only over the elements within that component. Section 7.2 describes component specifications further and defines when a component specification is true.

7.1.2 Checking Validity for Modular Proofs

A modular layout proof is true when each component specification P_c is true and when the proof is well-formed. This section describes an algorithm for establishing that a modular layout proof is well-formed; Section 7.2 describes how component specifications are checked to ensure that the overall proof is true.

Unlike component specifications, the well-formedness of a modular layout proof requires whole-page reasoning. It is thus essential that modular layout proofs avoid expensive reasoning about the browser layout algorithm. In a modular layout proof, the component specifications P_c already summarize the layout of each component. So, **no formalization of the browser layout algorithm is used to check the well-formedness of modular layout proofs**. Avoiding the need to reason about the entire page using the complex browser layout algorithm enables modular proofs to scale to much larger web pages than monolithic reasoning. This also makes it possible to combine multiple verification tools, each of which formalizes browser layout in its own way.

Checking the well-formedness of a modular layout proof means proving a Visual Logic implication. Visual Logic compiles to quantifier-free linear real arithmetic, so a solver such

as Z3 [DMB08] can be used to decide the truth of this implication. The challenge is finding an efficiently-solvable query equivalent to the implication to be tested. This query must express: (1) the structure of the page; (2) the component specifications P_c; and (3) the web page property Q, negated. If the query is satisfiable, there is a layout where the P_c hold but Q does not; otherwise, the P_c imply Q.

The main data type in the query is the *box*, which represents the size and position of an element on the page.[2] A box consists of an identifier and 28 real-number variables for its size and position.[3] The query declares two kinds of box variables: a "known box" for each box on the page and an "unknown box" for each free variable in Q; each unknown box is constrained to be one of the known boxes. Visual Logic expressions such as $b.\mathsf{top}[\mathsf{margin}]$ are compiled to formulas over these box variables: expressions in P_c over the known boxes, and expressions in Q over the unknown boxes. Visual Logic expressions can also refer to the parents, children, or siblings of a box. To support this, the query declares "pointer functions" from boxes to boxes and defines their values for the known boxes; these pointer functions define the structure of the page in the query. A selector test "$b \in \$(s)$" in Visual Logic is compiled to a disjunction "$b = \mathsf{box}_1 \vee b = \mathsf{box}_2 \vee \cdots$", where each box_i is a known box that matches the selector.

The resulting queries are efficiently solvable (see results in Section 9.3). The well-formedness of a modular layout proof is usually faster to check than its component specifications.

7.2 Component Specifications

Since a modular layout proof does not assume a particular formalization of the browser layout algorithm, different component specifications can be verified with different tools, including

[2]Some elements have multiple boxes (e.g., numbered lists), and some elements have no boxes (e.g., invisible elements).

[3]In total, 16 real numbers correspond to the sizes and positions of each of the margin, border, padding, and content areas of each box; plus, 12 real numbers represent its foreground and background colors, before and after gamma correction.

tools that formalize browser layout differently. This enables using different tools for different parts of the page as well as using different tools at different stages of the proof development process. This section describes four component specification verification tools and defines *independently-true* component specifications, which are true independently of the rest of the page and can thus be verified more efficiently.

7.2.1 True and Independently-true Component Specifications

A component specification abstracts over the layout of the elements in that component. Its pre-conditions and post-conditions state properties of the component's elements' layout that are relevant to proving a web page property.

Definition 7.4. *A component specification P_c of a component c on a page p is* ***true*** *if it is true of the elements in c for all possible layouts of the overall page p.*

For a modular layout proof to be true, its component specifications must be true. However, an element's layout depends on its context: the layout of the other components in the page. For example, the width of an element often depends on the width of its parent, and can even depend on subsequent HTML elements (for example, in the case of shrink-to-fit width). The height of a component often depends on the heights of its children. All of these elements may be part of other components. As a result, to test whether a component specification is true, one must reason about the whole web page. But such whole-page reasoning does not scale to larger web pages.

This motivates the identification of a stronger notion of truth which can be established by considering a component in isolation. This stronger notion is inspired by the insight that **a component specification that can be verified without consulting the rest of the web page must be true for all web pages that contain the component.** Such component specifications are called *independently-true*:

Definition 7.5. *A component specification P_c is* ***independently-true*** *of a component c on a page p if it is true on all web pages p' that contain c's HTML elements, and assign each*

element the same symbolic computed style as in p. Note that this requirement is vacuously true on pages p' that do not satisfy the preconditions in P_c.

One may think of a *true* component specification P_c as having an implicit precondition exactly describing the possible layouts of the rest of the specific page. Writing an *independently-true* component specification P_c replaces this complex and weighty implicit precondition with a simplified precondition sufficient to prove the property at hand.

7.2.2 Component Verification Tools

Troika currently supports four tools for verifying component specifications: random-test, model-check, whole-page, and component-smt; component specifications can also be "admit"ted. Some tools are sound, while others (random-test and admit) are not. Modular layout proofs are only sound when the component verification tools they use are sound; unsound tools are, however, useful during proof development. This subsection summarizes the random-test, model-check, and whole-page tools, which check whether a component specification is true, while Section 7.2.3 describes component-smt, which checks whether a component specification is independently-true.

Each tool takes as input the web page HTML and CSS,[4] the component c, the range of browser parameters, and the property to verify. The tool must either declare that the property holds, or produce output (such as a counterexample) to be shown to the Troika user.

random-test[n] and model-check random-test[n] and model-check are both based on the Cornipickle tool [HBGLB15]. random-test[n] compiles a specification to JavaScript and runs it in a remote-controlled instance of Firefox using n random sets of browser parameters. The model-check tool is similar, but exhaustively tests every possible setting of the browser

[4]The full web page's HTML and CSS are necessary for those tools that use whole-page reasoning (random-test, model-check, and whole-page) and thus need to compute layout and style information for elements outside the component.

parameters.[5] Both rely on Firefox's implementation of the browser layout algorithm. Thus, they can support components whose CSS is outside the subset formalized in SMT.

whole-page The whole-page verifier is based on VizAssert, and uses the Cassius framework to soundly verify component specifications. Although the assertion only refers to a single component of the page, whole-page still reasons about the whole page; as a result, the whole-page tool can verify any true component specification, provided it fits into the subset of browser layout formalized by Cassius.

7.2.3 Verifying Independently-True Component Specifications

The component-smt verification tool is a sound, SMT-based tool for verifying *independently-true* component specifications. component-smt's implementation is also based on Cassius, which verifies a whole-page layout property P by using an SMT solver to search for layouts of the page that: 1) are valid according to the browser layout algorithm, 2) fall within the specified range of browser parameters, and 3) do not match the property P. Solutions to this query are counterexamples to P. component-smt modifies Cassius to verify the layout of individual components by allowing the page's HTML tree to be partially undefined. These undefined portions correspond to the part of the web page outside the component; leaving them undefined allows the solver to synthesize that part of the page as part of its counterexample.

The main challenge in using partially-undefined HTML trees is that describing the layout of the elements in that partially-undefined portion would require the use of quantifiers and thus significantly slow down the solver. So, the component-smt tool instead uses the solver to search for the possible *effects* of elements outside the component on the layout of the component. By not explicitly representing those elements, component-smt thus avoids the need for quantifiers. To search for such effects, component-smt ensures that all constraints that relate

[5] The model-check only tests integer values of the browser parameters; non-integer values are rare in practice.

the layouts of two different elements pass through "pointer relations" such as "parent" or "first child". These relations are defined for pairs of elements within the component, but left undefined when they involve elements outside the component. As an example, if e_1 is inside the component, but its first child e_2 is outside it, the constraint $e_1.\mathsf{height} = 80 + e_2.\mathsf{height}$ is rewritten to $e_1.\mathsf{height} = 80 + \mathsf{first\text{-}child}(e_1).\mathsf{height}$, and $\mathsf{first\text{-}child}(e_1)$ is left undefined. The underlying solver then treats $\mathsf{first\text{-}child}(e_1).\mathsf{height}$ as an unknown real-valued variable and searches for possible heights that could cause e_1 to violate the component specification. Searching for the effects of the undefined portion of the HTML tree, instead of searching for that portion of the tree directly, is efficient enough to verify small components in seconds.

The component-smt tool inherits its soundness and completeness from the model of browser layout that it modifies: it only removes constraints and thus allows strictly more counterexamples, so soundness is maintained; and it only removes constraints on elements outside the component, preserving completeness (since independently-true component specifications cannot depend on the layout of those elements). We have not encountered any soundness or completeness problems in the current implementation of component-smt.

7.2.4 Visual Logic Extensions for Independently-True Specifications

An independently-true component specification must not reason about elements outside of the component, so its preconditions must express all effects of those elements' layouts. Preconditions on widths, heights, or positions can be expressed using ordinary visual-logic predicates; however, some CSS features allow distant elements to affect each other's layout. Troika extends Visual Logic with additional predicates that capture these long-distance interactions for use in preconditions. Note that such preconditions are only necessary to constrain the interaction of elements across components; component-smt directly verifies the interactions of elements within a component.

Margins In the browser layout algorithm, boxes have a *margin* where other elements are not supposed to be laid out. In some cases, margins of multiple boxes "collapse", or merge,

and predicates like non-negative-margins allow proof authors to control this multi-component interaction. These predicates are defined in terms of a new b.collapsed-margin field, which measures the size of a collapsed margin. This field allows defining the non-negative-margins(b) predicate, which is commonly used as a postcondition to prove that elements do not overlap.[6] In random-test and model-check, these fields are computed in JavaScript, while whole-page and component-smt use internal state in the Cassius browser layout algorithm that tracks this margin collapsing behavior.

Floating Layout To constrain how floating boxes from one component affect elements in another, Troika adds the starts-float-free(b, R) and ends-float-free(b, R) predicates (abbreviated sff and eff). Both predicates take a box b and a rectangle[7] R. The starts-float-free predicate asserts that floating boxes that precede b do not overlap[8] with R, while ends-float-free asserts no overlap for all boxes that either precede or descend from b. starts-float-free is usually used as a precondition in component specifications, while ends-float-free is used as a postcondition to prove starts-float-free predicates for later components.

The random-test and model-check tools check these predicates by examining all floating boxes in a given layout, while whole-page and component-smt translate starts-float-free and ends-float-free into constraints on the exclusion zones that Cassius uses to describe the position of floating boxes in the page. We have also developed helper functions to handle common uses of starts-float-free and ends-float-free. For example, no-floats-enter(b) generally asserts starts-float-free($b, [-\infty, b.\text{top}, +\infty, +\infty]$), that is, that preceding floating boxes are vertically above b, while float-flow-across(b_1, b_2) generally asserts that ends-float-free(b_1, R) implies starts-float-free(b_2, R). These helper functions concisely describe how floating boxes in one component affect elements in another component. More precisely, these helper func-

[6] Negative margins are legal in CSS and commonly used for tasks like centering. For these pages, proof authors can define alternative predicates in terms of b.collapsed-margin instead of using non-negative-margins.

[7] Defined by four numbers x_1, y_1, x_2, y_2, where each number is either real or $\pm\infty$.

[8] That is, that the margin areas of all boxes that precede b in an in-order traversal of the HTML tree do not overlap with b's margin area.

tions are defined as:

$$\begin{aligned}
\textsf{no-floats-enter}(b) &:= \textsf{sff}(b, [-\infty, b.\textsf{top}, +\infty, +\infty]) \\
\textsf{no-floats-exit}(b) &:= \textsf{eff}(b, [-\infty, b.\textsf{bottom}, +\infty, +\infty]) \\
\textsf{no-floats-exit}'(b) &:= \textbf{let } y = (\textbf{if } b.\textsf{height} = 0 \textbf{ then } b.\textsf{top}[\textsf{outer}] \textbf{ else } b.\textsf{bottom}) \textbf{ in} \\
&\quad\ \textsf{eff}(b, [-\infty, y, +\infty, +\infty]) \\
\textsf{float-flow-in}(b_1, b_2) &:= \forall R, \textsf{sff}(b_1, R) = \textsf{sff}(b_2, R) \wedge b_1.\textsf{top} \leq b_2.\textsf{top} \\
\textsf{float-flow-out}(b_1, b_2) &:= \forall R, \textsf{eff}(b_1, R) = \textsf{eff}(b_2, R) \wedge b_1.\textsf{bottom} \leq b_2.\textsf{bottom} \\
\textsf{float-flow-across}(b_1, b_2) &:= \forall R, \textsf{eff}(b_1, R) = \textsf{sff}(b_2, R) \wedge b_1.\textsf{bottom} \leq b_2.\textsf{top} \\
\textsf{float-flow-skip}(b) &:= \forall R, \textsf{eff}(b, R) = \textsf{sff}(b, R)
\end{aligned}$$

7.3 The Troika Proof Assistant

To make modular layout proofs accessible to proof engineers, Troika provides a convenient proof language for defining and checking modular layout proofs.

7.3.1 The Tactic Language

The Troika tactic language has two roles: defining web pages and theorems about them; and then proving these theorems by defining components and their specifications and indicating with which tools to verify them. Figure 7.1 gives a core grammar for this language.

Troika's "**page**" command loads web pages (from disk or via a URL), and its "**with**" block specifies ranges for each browser parameter, such as browser width and height, default font size, or any others supported by the verification tools used. The "**theorem**" command defines theorems about those pages in Visual Logic. The "**define**" command defines Visual Logic shorthands. Pages, theorems, and shorthands use separate namespaces. The "**proof**" command begins a proof and specifies the set of web pages for which to prove the theorem. A single proof can prove the same theorem across multiple similar web pages, as in Section 9.3.

$$\langle \textit{command} \rangle ::= \textbf{page}\ \langle \textit{page} \rangle = \textbf{load}\ \langle \textit{url} \rangle\ \textbf{with}\ \langle \textit{param} \rangle^*$$
$$\mid\ \textbf{theorem}\ \langle \textit{thm} \rangle = \langle \textit{assertion} \rangle$$
$$\mid\ \textbf{define}\ \langle \textit{fun} \rangle\ (\ \langle \textit{var} \rangle^*\ ) = \langle \textit{assertion} \rangle$$
$$\mid\ \textbf{proof of}\ \langle \textit{thm} \rangle\ \textbf{for}\ \langle \textit{page} \rangle^*\ \langle \textit{tactic} \rangle^*\ \textbf{qed}$$

$$\langle \textit{tactic} \rangle ::= \textbf{components}\ \langle \textit{cmp-name} \rangle = \$(\langle \textit{selector} \rangle)$$
$$\mid\ \textbf{for all}\ c \in \langle \textit{cmpset} \rangle\ \textbf{assert}\ \langle \textit{assertion} \rangle\ \textbf{by}\ \langle \textit{tool} \rangle$$
$$\mid\ \textbf{for all}\ c \in \langle \textit{cmpset} \rangle\ \textbf{require}\ \langle \textit{assertion} \rangle$$

$$\langle \textit{param} \rangle ::= \langle \textit{param-name} \rangle \in [\langle \textit{real} \rangle, \langle \textit{real} \rangle]$$

$$\langle \textit{cmpset} \rangle ::= \langle \textit{cmp-name} \rangle \mid \mathcal{C}$$
$$\mid\ \langle \textit{cmpset} \rangle \cup \langle \textit{cmpset} \rangle \mid \langle \textit{cmpset} \rangle \cap \langle \textit{cmpset} \rangle \mid \langle \textit{cmpset} \rangle \setminus \langle \textit{cmpset} \rangle$$

$$\langle \textit{tool} \rangle ::= \textsf{admit} \mid \textsf{random-test}[\langle \textit{num} \rangle] \mid \textsf{model-check} \mid \textsf{whole-page} \mid \textsf{component-smt}$$

Figure 7.1: The core syntax of the Troika tactic language.

Within a proof, each "**components**" tactic defines new components, identifying them using CSS selectors. Each element that matches that CSS selector becomes the root element of a component; a CSS selector can match multiple elements, so one "**components**" tactic defines a set of components. Defining multiple components with a single selector is important for partitioning large pages (as in Section 9.3).

The "**assert**" and "**require**" tactics define component specifications: a component with assertions (postconditions) A_i and preconditions R_j has the component specification $\left(\bigwedge_j R_j\right) \implies (\bigwedge_i A_i)$. Both "**assert**" and "**require**" operate on multiple components at once (using set operations on the sets of components defined by each "**components**" tactic and the set $\mathcal{C}$ of all components). In each assertion and precondition, the variable c is bound to the particular component, and can be omitted if not used. Troika also allows using "**for**" and "**component**", which are like "**for all**" and "**components**" but check that the component set contains exactly one component.

7.3.2 Implementation

Troika is open-source and freely available online.[9] The implementation includes an interpreter for the tactic language, a dispatcher for invoking verification tools, compilers from Visual Logic to SMT-LIB and JavaScript, and a core data structure for representing web pages and web page components. Troika uses the Cassius framework and Z3 SMT solver [DMB08] to check the well-formedness of proofs, as well as to execute the whole-page and component-smt tools.

To implement the random-test and model-check verification tools, Troika compiles the component specification to JavaScript and uses the Selenium library [Dav12] to run that JavaScript in a headless Firefox instance. Visual Logic expressions have straightforward JavaScript equivalents: Troika uses the CSS Object Model API to query element locations, hoists b.ancestor(P) expressions to recursive functions, and uses the Range API to access the

[9] At `https://cassius.uwplse.org/`

sizes of text boxes. However, on large pages, the straightforward JavaScript equivalent was often inefficient. For example, the Visual Logic assertion "$\forall b, b \in \$(\mathsf{a}) \implies \mathsf{onscreen}(b)$" quantifies over every box in the page, even if few boxes are generated by a elements. To fix this inefficiency, Troika searches the assertion for selector constraints on the quantified boxes and only tests the assertion on boxes matching that selector.

7.3.3 Parallelism and Caching

To check a modular layout proof, Troika must verify each component specification and check the well-formedness of the overall proof. The proof is valid only if each check succeeds. When a check fails and the verification tool produces a counterexample, that counterexample is shown to the user. Different components and verification tools run in parallel with each other and with proof checking, using a single-producer multi-consumer queue with a user-configurable number of parallel threads.

In order to speed up proof checking, Troika caches every invocation of a verification tool. These caches ensure that re-checking a proof is fast, which is convenient when adding additional theorems and proofs to a proof script. For simplicity, the cache requires an exact match for the web page HTML and CSS, component, browser parameters, and property being verified.

To make a match more likely, each verification tool can define an automatic pruning algorithm to discard the parts of the page it does not use. This allows component specifications to be reused across multiple web pages.

Of the four tools provided by the Troika prototype, only component-smt makes extensive use of pruning. Since component-smt does not rely on any part of the page outside the component, it prunes those parts of the page away, along with all CSS rules that do not apply to elements in the component and all fonts, classes, and IDs not referenced in the pruned CSS rules. Unnecessary rules, fonts, classes, and IDs can differ even on closely related pages; pruning them ensures that the cache key contains only information used by component-smt. Each pruning pass is conservative in order to preserve soundness.

Chapter 8
RELATED WORK

8.1 Accessibility of Web Pages

Title II of the American Disabilities Act and Section 508 of the Rehabilitation Act of 1973 both mandate nondiscrimination on the basis of ability for US government entities, and the Department of Justice has subsequently published guidelines for website accessibility [US 17a, US 17b]. Similar guidelines are expected for entities that are subject to Title III of the ADA [US 10]. Similarly, the European Commission issued the Web and Mobile Accessibility Directive [Eur16] to ensure public sector websites and mobile apps follow accessibility standards, and the proposed European Accessibility Act would extend these requirements to the private sector. In Japan, public sector websites have been held to accessibility standards through the Basic IT Law [Die00]. Similar standards exist in other countries as well.

These legal requirements, joined of course by moral considerations, have directed a lot of research in computer accessibility, including in the accessibility of user interfaces in general and web pages in particular. Major GUI frameworks, including the 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 [W3C16]. ARIA allows specifying the "role" of a web page element, so that accessibility tools can better describe it to a vision-impaired user, using the role attribute. For example, a link may be given a role of button, checkbox, link, menuitem, option, or others, reflecting the interaction model that link expects: clicking a link will navigate to a new page, for example, while clicking a checkbox will not. ARIA is important for JavaScript-heavy applications, where interacting with web page elements (like links) can trigger arbitrary behaviors. For example, for a link that will navigate to another web page,

describing the destination of the link is important, while for a checkbox it is important to describe its current state. ARIA also provides mechanisms for authors to describe events that change a web page (like new notifications arriving), to label states (like whether a checkbox is checked), and to define relationships between elements (like whether one controls the state of another). ARIA does not describe the visual appearance of a page; instead, it describes the semantic information that might be necessary for understanding and using a page. Unfortunately, ARIA use is spotty and often under-tested.

Accessibility guidelines Separately from ARIA, the W3C also publishes a set of guidelines for making web content accessible: the WCAG [W3C08]. The WCAG describes best practices for using ARIA, but additionally describes non-ARIA requirements, such as having captions for all images, transcriptions for all audio, and visible, high-contrast fonts. The WCAG is a document for web developers, and does not describe how its guidelines can be checked automatically. However, a wide variety of tools exist to help automate compliance with the WCAG and with the legal requirements described above. For example, Wave [Web17b], SOAtest [Par17], and 508checker [For14] are three tools that analyze web pages and offer warnings for elements that may be violating best practices such as avoiding justified text and having meaningful captions. These tools operate like linters: they are based on simple rules that match common errors that can be easily identified from web page source. For example, SOAtest offered suggestions to avoid justified text and pointed out the need to have captions for each image when run on some sample web pages. These tools do not analyze the visual layout of the web page, and thus cannot help with important guidelines dictating the size of text or the location of input elements.

Academic work has also been done on this problem. Raven [FS06] 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 to access accessibility information, such as content and role information for user interface elements [CYM11]. All of these tools only test visual

layout for particular browser parameters (or the equivalent for other GUI frameworks) so do not provide strong guarantees across the possible parameter values.

As expected, these automatic linters are not sufficient for producing accessible web pages. Mankoff, Fait, and Tran investigated different methods of discovering accessibility problems in web pages [MFT05], These authors compared four approaches to discovering accessibility problems, comparing each approach to a direct observation study of blind users completing a preset list of tasks:

- Review by expert web developers familiar with the WCAG guidelines
- Review by similar web developers with access to the JAWS screen reader
- Recommendations by the Bobby web accessibility testing tool
- Anonymous feedback from crowd-sourced screen reader users.

None of the four approaches discovered all of the accessibility problems, and each had a substantial false-positive rate that limited its utility; moreover expert review, especially with access to a screen reader, substantially outperformed the other approaches, with automated tools doing particularly poorly. Several of the problems missed by the automated tool required access to the visual appearance of the page. For example, the authors call out "poor formatting", "visual pairing", and "pop-ops" as visual properties not detected by automated tools. The limitations of current automatic accessibility checkers have filtered back into guidelines. For example, the WCAG [W3C08] suggests avoiding pixel sizes because they make it difficult to ensure the page will behave properly when text size changes, even though pixel sizes are necessary for designers to have exact control over web page appearance.

In fact, accessibility guidelines (such as the WCAG) recommend that web developers avoid common CSS features (such as pixel and percentage lengths) to avoid problems that are difficult to automatically test for. Of course, avoiding such features places limitations on the designs developers can achieve.

Accessibility properties Several recent papers have developed simple heuristics for detecting *layout failures*—web pages with problematic visual appearance. For example, Bigham's *opportunistic accessibility improvement* [Big14] proposes increasing the default text size on a web page until the size of the text introduces layout failures. Bigham's tool detects three types of layout failures:

- Scroll bars on elements that did not previously have scroll bars;
- Text overlapping that previously did not; and
- Very narrow lines of text (five words or fewer)

Also using simple visual heuristics, ReDeCheck [WKM17] attempts to detect pages that have layout failures when the screen size is changed. ReDeCheck measures the visual relationships between elements (such as alignment, horizontal or vertical adjacency, or containment) and detects screen sizes where these relationships change. Both of these tools work well despite their simple heuristics.

For more-complicated visual assertions, Cornipickle [HBGLB15] provides a declarative programming language for specifying desirable properties of web page layouts and checks them on a live browser as the user interacts with a web page. Cornipickle's language features first-order quantifiers, temporal modalities, and regular expression matches for text. This language makes it possible to express quite sophisticated layout problems: asserting that all elements of a list are aligned, ensuring that an element's position is not changed by user interactions, or ensuring that after logging in, the "log out" link is visible.

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. Cornipickle does not offer a verification tool like VizAssert for its assertion language; it simply offers a convenient method for testing assertions on particular browser parameters. VizAssert thus uses a similar language to Cornipickle, but it also verifies these properties (using Cassius), which allows it

to reason about all possible users, instead of specializing to a single user. (Troika's `random-test` is more similar to Cornipickle in this sense.) On the other hand, Visual Logic omits Cornipickle's temporal operators (since its existing uses in VizAssert and Troika do not involve reasoning about JavaScript).

8.2 Formalizing Web Page Layout

Developing a formal description of the browser layout algorithm follows in the footsteps of earlier efforts to apply formal reasoning to visual objects (including application GUIs and drawings) and earlier attempts to formalize portions of browser layout.

8.2.1 Formal visual reasoning

Several papers from the last few years have used formal methods to reason about graphical objects. A long line of work has examined using constraints to define the behavior of graphical user interfaces by constraining the positions of objects in the UI. Constraint-based graphics has been studied since Ivan Sutherland's Sketchpad system [Sut64] and used to lay out print [vW82], graphical user interfaces [ZM91, HM92], and web pages [BLM97]. Particularly applicable to Cassius is ConstraintSS [BBMS99], a recent effort directly applying the fruits of this literature to web pages. Also related to constraint-based visual reasoning is programming by manipulation [HBR14], which turns the constraint-based approach on its head by asking users to create visualizations by directly manipulating them; manipulations of the visualizations correspond to breaking and re-making the constraints that define it. Finally, Sketch-n-sketch [CHSA16] uses a similar approach—linking manipulations of graphics with manipulations of the programs that generate them—without the constraint-based underlying language.

ConstraintSS ConstraintSS [BBMS99] is a recent effort to combine constraint-based layout with the idioms of the web, such as stylesheets that can be used on many web pages and across different user preferences. ConstraintSS proposes allowing web page authors to write

constraints that will be used to determine the appearance of their web page. For example, the following constraint style sheet would ask the browser to use 13-pixel fonts for headings (`h1` elements) if such fonts are available or, if they are not, to use bigger fonts:

```
H1 { font-size: 13px }
H1 { constraint: font-size >= 13px }
```

The user would also be able constrain multiple elements in relation to one another, as in the following requirement that the elements `c1` and `c2` have the same widths:

```
@constraint #c1[width] = #c2[width]
```

ConstraintSS combines constraint solving with existing CSS idioms. Constraints in ConstraintSS can use selectors to choose elements to apply constraints to. A single ConstraintSS stylesheet can be used on multiple web pages. And at its core, ConstraintSS models web pages as document trees, unlike many constraint-based layout tools where screen objects are not organized in any higher structure. Using a tree structure for web page elements also allows ConstraintSS to use inheritance, much like CSS.

ConstraintSS must overcome two challenges of constraint-based layout: conflicting constraints and ambiguous layouts. General constraint languages allow writing multiple constraints that are not mutually satisfiable, such as requiring a certain element be both 12 and 15 pixels tall. It is also possible for two constraints to be mutually satisfiable for some screen sizes but not others; imagine requiring that an element both take up at least half the width of the screen, and be 400 pixels wide. ConstraintSS follows previous work in resolving these situations by annotating constraints with priorities, thereby establishing a hierarchy of constraints. When two constraints conflict, the lower-priority constraint is ignored. Priority systems give a well-defined meaning to conflicting constraints, but make predicting the behavior of the system more complex to reason about: only the highest-priority constraints can be assumed to hold. For web pages, priority systems are more natural than in other domains, since CSS already includes several priority mechanisms: user styles override author styles

override browser styles, `!important` rules override rules without that qualifier, and specific selectors override more general ones.

ConstraintSS must also overcome the challenge of ambiguous layouts: the given constraints may not uniquely determine the size and position of every element of the page. For example, the constraint `font-size >= 13px` allows both 13 and 14 pixel fonts if no other constraints are given. ConstraintSS leaves this question to the solver, which is tasked with finding a layout that satisfies the constraints. ConstraintSS uses Cassowary [BBS01] as the solver, which attempts to minimize the product of the "error" for each linear constraint (for `font-size >= 13px`, the error is `13px - font-size`) multiplied by a representation of the priority of that constraint. ConstraintSS also adds several modifications to the constraint solver, such as read-only variables, that make the constraints more predictable and easier to debug. Other constraint-based layout tools may use different solvers and different objective functions; for example, the Auckland layout editor for UIs [ZLSW13] uses the square of the error for each constraint, which ensures that the constraints have a unique best solution.

Constraint-based layouts are a powerful abstraction of layout, but come with challenges (conflicts and ambiguity). Resolving these challenges is possible (such as with priorities and objective functions) but also obscures some of the benefits (like ease of analysis and predictability) of constraint-based layout.

Like ConstraintSS, Cassius models CSS as a set of constraints that are solved to lay out the page. However, Cassius does not replace CSS or modify existing browsers. Instead, it provides a mechanized semantics for CSS, and as such, it could potentially be used to compile constraint-based layout languages to CSS.

Programming by manipulation Programming by manipulation, introduced by T. Hottelier [HBR14], is a methodology for helping users write visualizations. Programming by manipulation presents the user with a visualization, and allows users to change it by highlighting aspects of the visualization that they dislike. The system then generates a new visualization with that aspect changed. For example, the user may turn two bar charts into

a single stacked bar chart by dragging a single bar from one chart until it rests atop a bar from the other chart. The programming by manipulation system uses this information to recompute the visualization and produce a single stacked bar chart.

Behind the scenes, programming by manipulation maintains constraints that define the layout of the visualization. Every manipulation removes all constraints incompatible with the new position or shape of the manipulated object; thus, the system of constraints can never have conflicting constraints. Meanwhile, ambiguous constraints allow the system to present the user with multiple possible visualizations to help explore possible visualization designs. This exploration is done by labeling layout elements whose position is not fixed by existing constraints with an open padlock icon. The programming by manipulation system also draws these elements' axes of freedom.

Because the user is actively interacting with the set of constraints to produce a static visualization (unlike ConstraintSS, where the user is writing constraints once, and then allowing them to run in many possible contexts), programming by manipulation can afford to use less complex constraints. All constraints are chosen from a set of hard-coded *traits*, which are sufficiently general to make possible a wide variety of visualization types. The constraints in these traits are non-directional (unlike constraints with read-only variables in ConstraintSS) and are not prioritized (since constraints never conflict). The solver used is an SMT solver, which allows complex radial layouts and polynomial constraints.

Programming by manipulation is far from the only approach to allowing users to generate constraint-based layouts; most commercial editors for graphical user interfaces have such capabilities (often not phrased as editing constraints), and other research efforts have also adopted the strategy of limiting user interaction with constraints to avoid conflict or ambiguity [ZLSW13]; programming by manipulation is, however, relatively unusual in its focus on visualizations. There is a substantial literature on non-constraint-based languages for visualizations, such as the grammar of graphics [Wil05] implemented in popular libraries such as ggplot2 [Wic09]. Our work is complementary to much of this literature. However, such languages could have useful synergies with Troika: components generated from such

languages could automatically provide useful preconditions, making their use within Troika easier.

Sketch-n-sketch Programming by manipulation and similar efforts can be seen as linking graphical program output with program text (where the program is the set of constraints, including their priorities and directionality where applicable), so that, just as manipulating the program manipulates its output, manipulating the output would manipulate the program. The benefit of this bidirectionality is that visual properties are often much easier to describe by direct manipulation then by describing the program that produces the right output. Sketch-n-sketch [CHSA16] is the most complete example of this vision, which allows the user to write arbitrary programs in a functional subset of Scheme, and link manipulations of the graphical output of these programs to manipulations of the program text itself.

Unlike ConstraintSS and Programming by Manipulation, which formalize the semantics of layouts as a set of constraints, Sketch-n-sketch formalizes layouts as the output of a arbitrary program in a Turing-complete language that produces a collection of graphical objects (rectangles, say) as output. The key challenge to graphics generated by arbitrary programs is handling control flow—it is possible for similar outputs to be produced by dramatically different computations. The authors of Sketch-n-sketch note that for most graphics, this sort of confluence is unusual because control flow is usually simple and driven by data. Instead of control flow, the visual appearance of the graphic is usually derived from constants embedded in the source code.

Sketch-n-sketch uses this insight to quickly translate between output manipulations and program manipulations. As it executes a graphics program, Sketch-n-sketch stores a trace of arithmetic operations used to compute the position and size of every object on the screen, and a path condition for the control flow path followed. When the user manipulates an object, Sketch-n-sketch uses the trace to solve for values of constants that result in the new position, and then verifies that these new values also satisfy the path condition. In other words, Sketch-n-sketch synthesizes new values of constants under the assumption that

control flow will not change, and separately verifies the assumption once new values are chosen. This separation means synthesizing new values of constants can use simple solvers for arithmetic computations instead of more complex solvers for boolean logic, resulting in near-instantaneous synthesis of new constants.

8.2.2 CSS formalization

Other researchers have formalized parts of the browser layout algorithm and mechanically reasoned about it. The most prominent such work is pbrowser [MB10], which formalizes part of the browser layout algorithm using attribute grammars. This specification is then used to generate parallel execution schedules for web page layout. For CSS selectors in particular, a major focus of pbrowser, an alternate approach uses tree logics [HLO14]. We discuss both.

pbrowser pbrowser formalizes the browser layout algorithm to derive fast parallel browser implementations. This requires specifying the algorithm in a form that does not fix an evaluation order; attribute grammars to fill this role perfectly. An attribute grammar combines constraints with a tree structure[1] by specifying per-node properties and a set of constraints that are true of every node which must be expressed in terms of that node's properties and the properties of its immediate children, parent, and siblings. The attribute grammar structure is appropriate for an algorithm, like browser layout, that operates uniformly on each node, but the restriction to interacting only with immediately adjacent nodes complicates the handling of several features, most notably floats.

For example, a layout algorithm could compute the y-positions of elements like so:[2]

$$b.\mathsf{y} = b.\mathsf{previous.y} + b.\mathsf{previous.height}$$

[1] Note that ConstraintSS also combines constraints with a tree structure; however ConstraintSS is interested in specifying different constraints for different tree nodes, while pbrowser represents a layout algorithm that operates uniformly on nodes.

[2] A full layout algorithm would have to handle elements without previous siblings, would have to define how heights are computed, and much more.

pbrowser would then analyze this definition to derive a parallel implementation of layout; maybe be a parallel prefix sum of heights. Rephrased in terms of constraints, pbrowser analyzes constraints to synthesize a fast parallel solver for those constraints.

pbrowser's focus on speed also suggests a focus on selector matching, which takes up a plurality of layout time on many large web pages. They use standard optimizations, including bottom-up selector matching, caching of small selectors, and parallel tiled traversals of the tree of elements, and find that these optimizations yield large speed-ups.

Due to limitations of attribute grammars, these efforts cover a smaller subset of CSS than Cassius. Furthermore, these formalizations are not meant for verification, only layout, and thus can not be used for mechanical reasoning.

Hague, Lin, and Ong An alternate approach to formalizing selectors is presented by Hague, Lin, and Ong [HLO14], who use tree logics to describe selectors. Hague, Lin, and Ong focus on detecting redundant CSS selectors: CSS selectors that are never the most specific selector matching any element. Though this question is easy to answer for a fixed element tree, Hague, Lin, and Ong aim to answer it for a page that may be modified by JavaScript that responds to user interactions, so that a CSS selector is redundant only if it is never the most specific selector matching an element, *no matter how a user interacts with the page*. Their representation of selectors by tree logics enables them to solve this problem.

A tree logic is a modal logic whose Kripke semantics has the set of tree nodes as the set of worlds and modal operators corresponding to moving down or up the tree by one or many steps. For example, the formula

$$P \wedge \langle\uparrow\rangle(Q \wedge \langle\downarrow^*\rangle R)$$

is true at all nodes where P is true of that node, Q is true of its parent node, and R is true of all descendants of its parent node: the modal operator $\langle\uparrow\rangle$ relates each node to its parent node, while the modal operator $\langle\downarrow^*\rangle$ relates a node to all its descendants.

The key benefit of using tree logics to formalize CSS selectors is that facts can be derived

in the tree logic without reference to an explicit element tree. This allows Hague, Lin, and Ong to ask whether two CSS selectors match the same elements on all of the possibly-infinite set of element trees that can possibly be produced by a set of JavaScript tree rewriting steps. They prove this question EXP-complete, along with some hardness results for special cases, and describe an encoding in pushdown automata, which allows using existing solvers to answer such queries.

8.3 Web Developer Tools

There is a long line of work developing tools to help web developers author, test, and debug web pages and web applications. We cover testing tools (for interactive applications), and authoring tools (for writing HTML and CSS).

Testing interactive applications The need for tools that understand visual appearance has spawned several academic tools. We focus on Sikuli [CYM10], a domain-specific language that allows interacting with an interface and comparing its appearance to known-good appearances. The Sikuli workflow involves application developers writing application-specific interaction scripts that can then be automatically run (by, say, a continuous integration server) to make sure assertions in the script are not violated and regressions do not occur. For example, a script may click through a series of dialogs, ensuring that each dialog had the expected information and options (see Figure 8.1). To make writing and maintaining these scripts simple, even for users without programming expertise, Sikuli uses pixel-for-pixel comparisons of portions of the screen both to describe on-screen objects (such as buttons to click) and in its assertions (to describe how a portion of the screen must look). Using pictures makes it easy to understand the script, and provides a measure of independence from the application source code.

To apply Sikuli to accessibility testing, the authors combine this pixel-by-pixel information with accessibility information provided by many application frameworks [CYM11]. In this extended Pixel+Accessibility Framework (PAX), scripts may locate on-screen objects

Figure 8.1: An example Sikuli script, which changes the preference for "icons and text" or "icons only" for menu buttons and tests that applying the change correctly changes the behavior of the user interface. Note the use of bitmap images to describe portions of the user interface, both to drive the script (for example, to identify the button to click) and to describe assertions. The PAX extensions to Sikuli allow additionally using accessibility information (such as operating system role annotations) to identify screen objects.

either by bitmaps or by accessibility information such as widget roles, and may also access widget states provided for accessibility. Such a combined script would be able to check, for example, that the text size matches the value of a slider. Sikuli has been applied to web applications [LSRT15], but the biggest advantage of pixel-based testing is that pixel-based testing does not require the applications being tested to use a particular GUI framework. On the web, widespread use of CSS selectors makes addressing individual elements easier, and all applications use the same GUI framework: HTML and CSS. Alternatives to Sikuli scripts are thus less onerous on the web, and Sikuli's advantages less notable.

Since writing such scripts may be challenging for users without programming experience, several research teams have investigated methods for automatically generating such scripts from user interactions [OAFG98, MBN03, LSRT15] A key challenge in these tools is mapping between a high-level understanding of the application or web page (say, whether a user is logged in or not) and a low-level understanding (say, whether there is a "Log out" button).

This mapping must be carried out both for the actions that drive the script and the assertions that are tested. While Sikuli and other visual testing tools aim at different problems than VizAssert (they test simple predicates of interactive applications, instead of complex predicates of static pages), a similar high-level / low-level split exists for VizAssert: high-level intuitions such as "text should not overlap" have to be translated to low-level assertions about the position of HTML elements. A common theme from such work is the importance of developing the right formal descriptions of correct graphical interface behavior [XM07]. Visual Logic incorporates this insight by developing an expressive logic in which developers can specify their accessibility and usability properties.

Among practitioners, a large ecosystem of tools exists for designers to test their pages against specific instances of browser, operating system, and user parameters [Bro18a, Bro18c, Bro18b]. Such tools load pages in browsers running in virtual machine instances, returning screenshots to designers so they can manually check against expected layouts. This testing approach is easy to use and widely adopted, but requires manual inspection and like heuristic tools does not provide guarantees.

CSS authoring tools CSS authoring tools have focused on transferring CSS styles from one page to another. We discuss Bricolage [KTAK11], which uses heuristics to transfer the design of one web page to another.[3]

Bricolage is possible thanks to the the design of CSS: applying CSS properties from one element to another frequently makes the elements look similar, without requiring changes to the property values. Thus, translating styles mainly depends on identifying matching elements on the two pages; Bricolage accomplishes this by segmenting the page into independent components, with the segmentation guided by the visual appearance of the page. Then, segments are matched between the source and destination page using a graph optimization algorithm the objective function for which is learned from user examples. Once

[3] Webzeitgeist [KST+13] uses similar techniques to organize a corpus of common design elements across web pages.

elements are matched, CSS selectors for elements on the source page are translated into selectors for matching elements on the destination page.

Bricolage uses a key duality of web pages: web pages have both semantic structure (the tree structure of the HTML) as well as the visual structure (driven by the position and size of laid out elements). Bricolage's graph optimization algorithm balances visual and semantic concerns: the objective function optimizes the sum of three terms, one of which is the visual information learned from user examples and the other two incentivizing the maintenance of ancestry and sibling relationships between matched elements. Some of the assertions we formalize in Visual Logic play on the same duality. For example, the assertion that ensures that higher-level headers are also bigger relates the semantic concerns of heading levels with the visual concerns of text size. More generally, a key accessibility requirement is that sighted users, who gather information primarily visually, and vision-impaired users using a screen reader, whose information is gathered primarily from semantic information embedded in the HTML (possibly including ARIA role information), must receive the same understanding of the web page. Tools that understand the visual layout of the web page would allow developers to test that the visual and the semantic information match.

Other tools help developers understand how changes to CSS affect the visual appearance of a web page. For example, SeeSS [LKL+13] automatically tracks changes in web page layout due to changes in CSS, to help developers avoid introducing bugs as they modify CSS code. SeeSS highlights areas of web pages that changed between CSS revisions and makes it easy for developers to review these changes across a large collection of web pages. The technique uses a hard-coded list of which properties to modify given a course-grained description of the layout problems as its only guiding principle, and seems to ably fix simple layout problems. VizAssert and Troika provide automation facilities that complement these efforts, and investigating avenues for integration is an interesting direction for future work.

Many authors have developed heuristic debugging tools to help web developers find, debug, and fix layout problems in web pages. Several tools use a mix of heuristics and static analysis to identify layouts likely to contain errors [WMK15, WKM17]. Other tools attempt

to automatically fix errors [MAMH17, Big14, MAMH18b, MAMH18a], using heuristics to detect whether a fix preserves web page correctness. Other tools attempt to detect cross-browser differences for web pages, often using heuristics to build a high-level model of the page behavior [RCVO10, MP11, CPO12] These tools are useful, but are each specific to a single web page property. Finally, some tools help developers transfer styles or content between web pages [MCC12, MvC14]. These heuristic tools are useful, and the formal semantics of browser rendering developed in this work could be used to put these tools on a more rigorous footing.

HTML authoring tools Authoring tools for HTML focus on generating HTML from visual representations of the final web page. For example, Remaui [NC15] turns mock-ups of Android applications into Java code that builds a similar-looking user interface. Remaui uses machine vision and optical character recognition to identify screen objects such as text and buttons, and then uses the position of these objects, together with simple heuristics, to connect these screen objects into a hierarchy. Then, based on the hierarchy generated, Remaui chooses Android components to represent each node. For example, a node with many similar children may be instantiated in a ListView. The segmentation algorithm used in Bricolage, Bento, uses a similar approach [KTAK11]. Further afield from web development, ReVision [SKC^{+}11] applies machine vision to understand the structure of data visualizations, and synthesize new visualizations of the same data; similar tools [HA14] allow easy modification of data visualizations built with the D3 library. For these tools, an additional challenge is inferring the data used in generating the visualization.

8.4 Formal Methods

The large and dynamic formal methods research community has developed a variety of techniques to soundly prove that programs satisfy certain behaviors for all possible inputs. Cassius, VizAssert, and Troika develop this approach for the web.

8.4.1 Encoding languages to SMT

In recent years, many researchers have used reductions to SMT to efficiently solve verification and synthesis problems. A common paradigm in this work is the solver-aided language [TB13]: a programming language paired with an encoding of its semantics to SMT, so that mechanical reasoning about programs (including verification and synthesis queries) can be efficiently executed by SMT. Boogie [Lei08] serves as a back-end for general purpose program verification by using Z3 to prove verification conditions [Lei10]. Alive [LMNR15] and PEC [KTL09] verify compiler optimizations using an SMT solver. Batfish [FFP+15] verifies the dataplane for Datalog programs. In each case, verification conditions and program properties of a high-level language are compiled to efficiently-solvable SAT or SMT constraints. Cassius follows a similar architecture for CSS: queries about CSS stylesheets are compiled to SMT constraints and solved with Z3. Smten [UD14] and Rosette [TB14] are solver-aided *host languages*—they translate a programming language interpreter into a solver-aided language backend. Unlike these general-purpose languages, which aim at automating a wide spectrum of problems, Cassius focuses on providing both a declarative semantics for CSS and an efficient host platform for reasoning about web page layout. Cassius is unusual in that it offers solver aid for an existing language, instead of designing a new language specifically to allow the use of solvers.

Modular Verification Modular verification techniques for model checking [GL94] pioneered assume-guarantee-style reasoning [Sta85] to make it possible to summarize component behavior and relationships between components in a larger system. Such techniques have been applied in a number of tools, including MAGIC [CCG+03] for semi-automatic verification of C programs and Dafny [Lei10], a programming language with built-in constructs to ease static verification. Similar reasoning has been scaled up to verify the 60,000 lines of code implementing the Microsoft Hyper-V hypervisor [DMS+09]. One of the major domains for modular verification is for security [App16], including using finite-state models to modularly verify data transfer protocols [HO83], and the correctness of com-

pilers [Ler06, BDL06]. Additionally, numerous program logics have been developed to modularly decompose properties of programs, especially in the domain of shared-memory concurrency [TTA+13, DYDG+10, NLWSD14, TVD14, RVG15, SB14], and techniques have also been developed for composing proofs in general [JSS+15, JKBD16].

Troika differs from all of these previous efforts due to targeting web pages, where traditional programming techniques cannot easily be applied. Compared to domains studied in past research, web pages introduce many new challenges: there are no clear computational units like functions that define module boundaries; there is no "heap" or other key stateful structure that existing techniques can be applied to abstract over; components within a web page tend to carry fine-grained context dependencies and often impact the layout of subsequent components in the page; and there is not a clear time order to web page layout, so it is not obvious how pre- and post-conditions can be used to summarize the layout of components. Troika addresses these challenges and pioneers new techniques for bringing modular verification to web pages.

In addition to building on SMT solvers like Z3 [DMB08], Troika also mirrors some aspects of such solvers' architectures. Troika composes multiple tools for reasoning about web page components in a general logic. As in SMT, this design allows analysis tools to cooperate when verifying complex inputs without requiring tight coupling between the analyzers. Having laid this groundwork, in the future we hope to see more research analyses adopt Troika's tool interface so that a broader range of properties and pages can be effectively verified.

Chapter 9

EVALUATION

To evaluate the formal verification approach to web page layout properties, we tested Visual Logic, Cassius, VizAssert, and Troika on three suites of web pages: the official W3C conformance tests, which we use to ensure Cassius faithfully formalizes the semantics of browser rendering; 51 templates from the design forum `freewebsitetemplates.com`, which ensure that VizAssert is a practical verification tool for realistic web pages; and several pages from the "Joel on Software" blog, which show that Troika allows scaling layout verification to pages 11× larger than the monolithic VizAssert approach can support.

9.1 CSS Working Group Unit Tests

This section demonstrates that the Cassius semantics faithfully models the CSS standard and that its encoding to LRA is efficiently-solvable by comparing it to Mozilla Firefox on a set of 3651 standard browser conformance tests [CSS11].

We ran two experiments. In the first experiment, we checked that the Cassius semantics for CSS accepts the layout produced by Firefox, demonstrating that this semantics is *sufficiently weak* (with respect to Firefox). Our results show that Cassius agrees with Firefox on all but 15 tests, on which Cassius produces correct layouts according to the standard, while Firefox produces slightly different layouts due to rounding errors or known bugs. In the second experiment, we checked that the Cassius semantics for CSS rejects other layouts by using a form of solver-aided mutation testing. Only 0.7% of mutants are accepted by Cassius, largely due to font metrics, which Cassius does not model. These results show that the Cassius semantics is *sufficiently strong* in practice.

9.1.1 Methodology

The official conformance tests [CSS11] measure the interoperability and correctness of CSS layout implementations. Tests exist for every section of the CSS standard, including many aspects of CSS layout (such as fonts, tables, and print media) not described by Cassius. The Cassius fragment of the standard is covered by 3651 conformance tests. Each test (e.g., Figure 9.1) consists of a web page with an English-language description of the expected output. Tests also have an associated reference page that achieves the expected layout using a simpler stylesheet. The tests are small, rarely consisting of more than a dozen elements and a few stylesheet rules. Furthermore, the W3C maintains a public website through which volunteers manually compare a browser's layout of the test to the instructions and to the reference page.

We applied Cassius to all 3651 conformance tests that test sections of the standard supported by Cassius. To check that it produces the expected layout, we use the Mozilla Firefox 60.6.1 browser as a test oracle, since volunteers have already checked that Firefox passes these tests. For each test, we employ a script (based on the CSSOM JavaScript API) to extract the layout generated by Firefox. This layout is used as the basis for acceptance and rejection tests.

9.1.2 Acceptance Tests

Acceptance testing ensures that the Cassius semantics allow the rendering produced by Firefox. To do so, these tests feed Cassius the test's document tree and stylesheet, as well as the concrete rendering generated by Firefox, and check that Cassius allows the given layout. Note that such a check exercises our encoding even with fully concrete inputs: Cassius uses the solver to search for a trace of its non-deterministic layout algorithm that admits those inputs.

Test passes if there is space between the blue and orange lines.

Filler Text

Figure 9.1: A test from the W3C CSS 2.1 conformance tests; this one is named padding-left-applies-to-008. Tests pass or fail based on the English-language instructions provided on each page or by comparison to a reference page that achieves the desired layout using a different, simpler stylesheet.

Table 9.1: The results of applying Cassius to the W3C test suite. The "Accept" and "Reject" columns give the number of tests on which Cassius does and does not accept Firefox's layout. See the text for a description of the cases where Firefox and Cassius disagree. The "Unsupported" column counts unit tests in that section using CSS features outside the subset formalized in Cassius.

	Test group	Accept	Reject	Unsupported
§8.1	Box dimensions	5		
§8.3	Margins	318		25
§8.3.1	Collapsing margins	129	1	3
§8.4	Padding	380		12
§8.5.1	Border width	350	10	11
§8.5.2	Border color	797		3
§8.5.3	Border style	146		
§8.5.4	Border interactions	187		1
§8.6	Inline	3		9
§9.1.1	Viewport	2		2

§9.1.2	Containing blocks	4		1
§9.2.1	Block boxes	9		
§9.2.2	Inline boxes	13		1
§9.2.2.1	Anonymous inlines	1		
§9.3	Positioning	25		3
§9.3.1	Position types	26		
§9.4.1	Block formatting	23		
§9.4.2	Inline formatting	31		3
§9.4.3	Relative positions	10		4
§9.5	Floats	185	4	15
§9.5.1	Positioning floats	24		3
§9.5.2	Clearance	60		9
§9.6	Absolute position	10		
§9.7	Display/position/float	34		
§10.1	Containing block	56		4
§10.2	Width	105		3
§10.3.1	Inline non-replaced	3		
§10.3.2	Inline replaced	19		
§10.3.3	Block non-replaced	10		2
§10.3.4	Block replaced	4		
§10.3.5	Floating non-replaced	10		2
§10.3.6	Floating replaced	9		
§10.3.7	Positioned non-replaced	31		6
§10.3.8	Positioned replaced	25		5
§10.3.10	Inline-block replaced	4		
§10.5	Height	101		6
§10.6	Calculating heights	2		

§10.6.1	Inline non-replaced	3		
§10.6.2	Inline replaced	40		
§10.6.3	Block non-replaced	23		1
§10.6.5	Positioned replaced	39		
§10.6.6	Complicated cases	4		
§10.6.7	Auto heights	4		
§10.8	Line height	7		3
§10.8.1	Leading	192		27
	Total	3463	15	164

Results The results are shown in Table 9.1. The tests are grouped by the section of the standard that they cover. Tests were run on an Intel Core i7-4790K quad-core CPU, with a version of Z3 built from the `opt` branch on 11 March 2015. Of the 3651 tests, Cassius agrees with Firefox on all but six; the disagreements are described below. The full suite of tests took 178 minutes to run, for an average of less than four seconds per test. Note in particular 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) of the standard, which describe CSS features that require finitization reductions to properly support.

Disagreements Between Cassius and Firefox Cassius passes all but 164 tests; all 164 of these failing tests use unsupported features, most prominently vertical alignment, right-to-left text, and SVGs, and thus are out of scope for the Cassius semantics. Among the passing tests are 15 for which Cassius's layout is different from Mozilla Firefox's. On 11 tests, the cause of the disagreement is rounding error within Firefox, and Cassius's layout is correct according to the CSS standard. On the other 5 Firefox's behavior in these cases is known to be incorrect [Web17a]; we manually confirmed that Cassius's layout matches the reference layout.

There are two sources of rounding error in Firefox. First, Firefox represents on-screen

lengths as fixed-point values rounded to a sixtieth of a pixel. This accuracy is usually sufficient to render web pages properly, since errors of sixtieths of a pixel are not visible. [1] In some cases, the true position of a point on the screen is not an exact multiple of a sixtieth of a pixel; for example, the test margin-collapse-032 creates a box whose padding is 2% of 1898 pixels, that is, $37.96 = 2277.6/60$ pixels. Firefox rounds the padding of the box to 37.95 pixels. Since Cassius uses real (infinite-precision) arithmetic, per the CSS standard, its specification does not admit this rounded value, thus disagreeing with Firefox. Second, Firefox rounds border widths and text positions to the nearest pixel, to ensure that text can be properly hinted and that borders appear to have the same width on both sides of any box. Modifying each test to correct the rounding errors results in a passed test.

9.1.3 Rejection Tests

The second experiment checked that Cassius rejects invalid layouts and was run during the development of Cassius. Since it was run on an older version of Cassius, its results are not directly applicable to Cassius's current form, but it does show that it is possible to achieve a conservative formalization of browser layout without permitting behaviors that mainstream browsers do not exhibit. In the second experiment, we mutated a core suite of 2075 acceptance tests as follows. For each test, we randomly chose a block box's width, height, or x or y position, and replaced it with a hole. Cassius was then asked to find a layout of the page that differs from Firefox's in the chosen measurement. Finding such a layout represents a failure—a test case on which the Cassius semantics is weaker than that of Firefox. This procedure was repeated ten times for each test in the test suite, resulting in 20750 total attempts to find an invalid layout accepted by Cassius.

---

[1] Note that "pixel" here refers to CSS pixels, which may not be real pixels; CSS pixels are an integer number of real pixels so that an inch is as close to 96 CSS pixels as possible. On high-resolution devices such as phones, Firefox may only represent lengths to an accuracy of one fifteenth of a pixel. Firefox has used one-sixtieth of a pixel as the basic unit of length since 2008 [Moz15]; before then Firefox used a twentieth of a point, or a fifteenth of a pixel, as the base unit of layout [Lod08]. WebKit also uses fixed-point values, but in terms of sixty-fourths of a pixel [Buj15]; it used sixtieths of a pixel until 2012, and switched in order to reduce precision loss when converting for floating point for use by JavaScript [Ekl12]. Both layout engines used to use floating point, but switched to avoid rounding errors [lev12].

Results Of the 20750 mutants, Cassius accepted only 152, for an overall success rate of 99.3%. We manually investigated every accepted mutant, and found that they were all accepted due to two causes. The majority of the mutants (126 out of 152) were accepted because Cassius does not model font rendering. In particular, Cassius accepts layouts that subtly move the boundary between two lines of text. The remaining mutants (26 out of 152) are due to non-determinism in the CSS 2.1 algorithm for computing the shrink-to-fit width of floating boxes. Cassius treats the non-determinism in the specification by accepting several layouts, allowing Cassius to sometimes find layouts different from the one found by Firefox. Several drafts for the CSS 3 standard fully specify the shrink-to-fit algorithm, so Cassius could be updated to remove this non-determinism once the draft standards are accepted.

9.2 Free Website Templates

To test that VizAssert can verify that realistic web pages satisfy common accessibility and usability guidelines, we evaluated whether 51 web pages satisfy the 8 general-purpose assertions described in Section 4.3. We also ran VizAssert on the 6 page-specific assertions and their corresponding web pages.

9.2.1 Subject Web Pages

We collected 51 web pages from an online community of web design professionals that nominates, selects, and publishes high-quality web pages: FreeWebsiteTemplates (FWT) [Fre17]. The published web pages are written by different developers, so they cover a cross-section of common web design techniques. See Figure 9.2 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 51 web pages that fit within the subset of CSS supported by Cassius.

	rehabilitation-yoga	tourism-surfing	amusement-park
Elements	36	48	87
Boxes	72	112	180
CSS Rules	239	232	266
Fonts	8	9	8

Figure 9.2: Three pages from the FreeWebsiteTemplates suite plus statistics about their sizes. In each screenshot, the dashed lines (black or white) are overlaid on the page to show the component boundaries used in Troika proofs in Section 9.2.3.

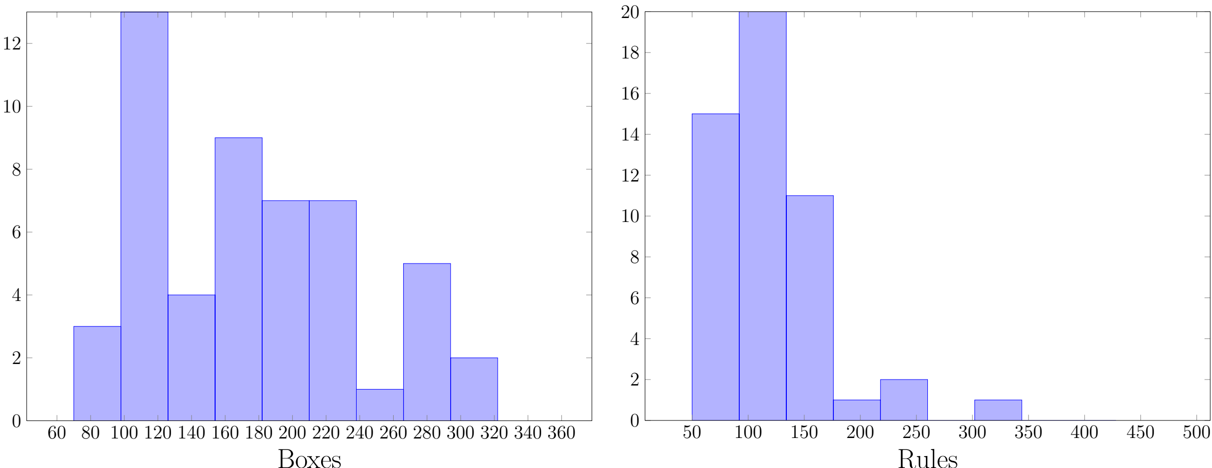


Figure 9.3: Histogram of the number of boxes and CSS rules for the 51 web pages.

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 51 web pages average 78 elements (56–93 Inter-Quartile Range), 176 boxes (119–222 IQR), and 128 rules (89–145 IQR). See Figure 9.3 for histograms of these statistics.

9.2.2 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 of 1024–1920 pixels, screen heights of 800–1080 pixels, and default font sizes of 16–32 pixels. In total, there were 349 successful verifications, 38 true positives, 21 false positives, and 6 timeouts (see Table 9.2).

VizAssert outputs browser 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 9.4). The assertion requires more important headings to be larger, but the section titles in the page's sidebar (in second-level `h2` headings) are rendered in `14px` type, while sidebar links

Table 9.2: Results of verifying accessibility assertions for 51 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.

#	Assertions	Verified	True positive	False positive	Timeout
1	text_size	41	9		1
2	interactive_onscreen	44	1	6	
3	line_width	40	9		2
4	accessible_offscreen	51			
5	no_horizontal_scroll	51			
6	heading_size	30	17	2	2
7	overlapping_text	37		13	1
8	line_spacing	50	1		
9	contrast	1			
10	no_text_on_picture	1			
11	dropdown_offscreen	1			
12	three_column	1			
13	visible_text		1		
14	button_large	1			
Total		349	38	21	6

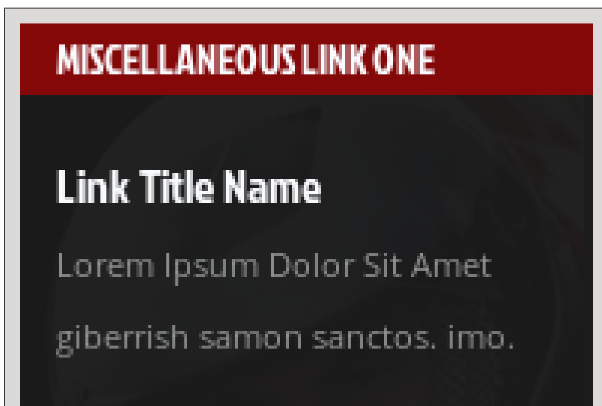


Figure 9.4: On the left, a true positive discovered by VizAssert: the section title, "Miscellaneous Link One", is set in smaller type than subheadings. On the right, a false positive: VizAssert believes that "SALE" can overlap with the date, because it does not reason about the lack of descenders in the letters of "SALE".

(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 9.4), 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 formalized. Cassius 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 9.5, due to the complexity of the browser layout algorithm. Instance size is largely independent of the assertion being verified, instead depending mostly on the size of the page, since the assertions are small relative to VizAssert's formalization of browser layout. 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 6 executions out of 414 (2.2%) timed out in 30 minutes. Figure 9.5 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 (see Assertion #3 in Section 4.3).

9.2.3 Modular Layout Proofs

This section compares the experience of using Troika and VizAssert. To do so, this section defines a systematic approach to constructing Troika proofs and demonstrates it on 8 proofs from the FWT suite. Statistics on these proofs can be found in Table 9.3. Every proof uses only the `component-smt` tool, though other tools were used during proof development. Overall, we find that the proofs are short (15 lines average), that proofs of similar properties or similar pages are similar (reducing the burden of proof development), and that modular layout proofs provide significant advantages over VizAssert, even for pages of the scale VizAssert is designed for (including a speedup of 1.8–64 $\times$).

9.2.4 Writing Modular Layout Proofs

A systematic proof development strategy directs the proof author's efforts so that each step makes progress toward the overall proof goal. For Troika, an effective proof development strategy proceeds in three phases:

1. The proof author decomposes the page by visual inspection, producing components such as headers, footers, sidebars, and body text. If the theorem focuses on a particular page element, that element is encapsulated in a component.

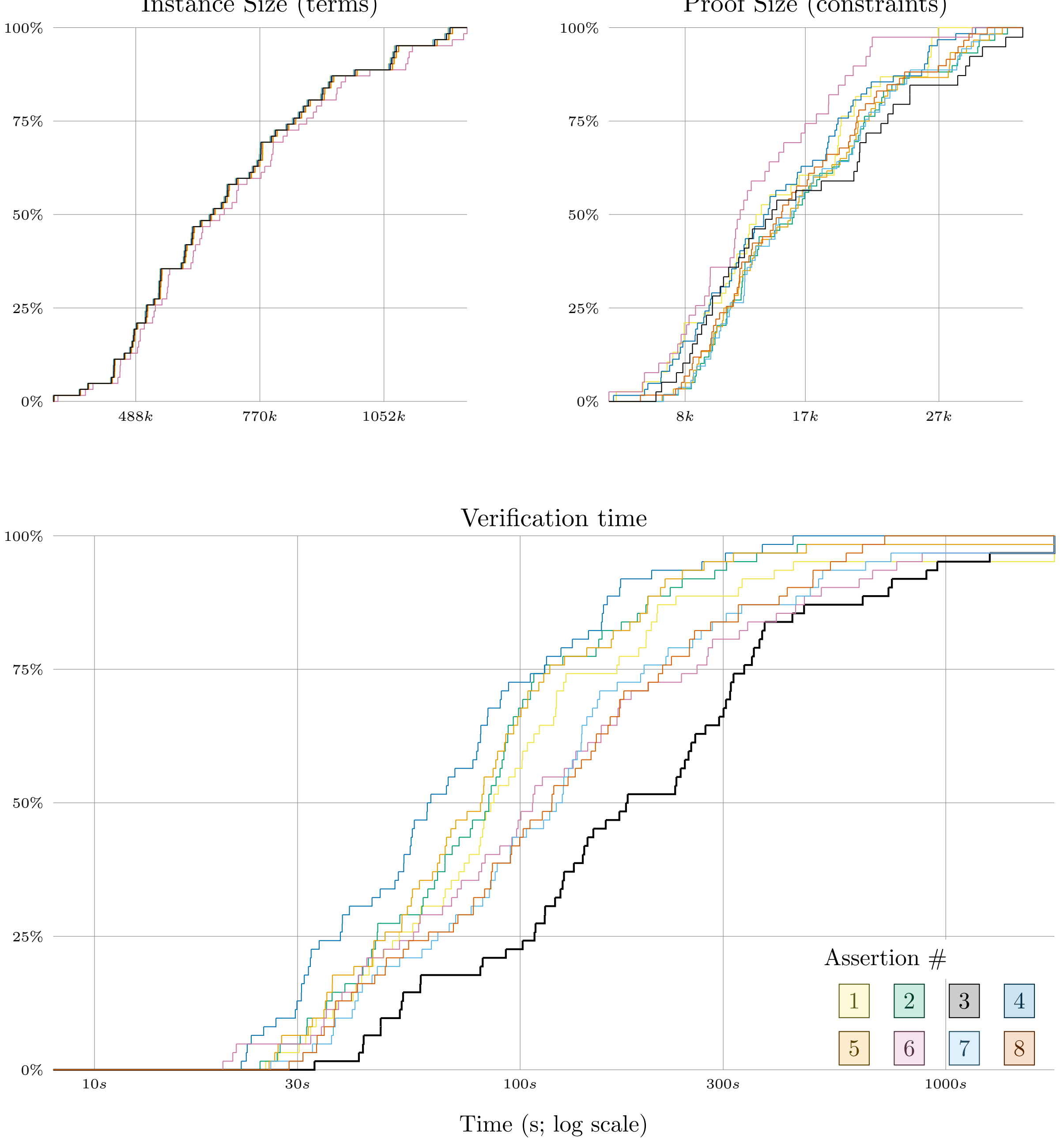


Figure 9.5: CDF of SMT instance and unsat core for each assertion tested by VizAssert, and the time required to check satisfiability for the SMT instances. Assertion #3 is drawn thicker for users who printed in black and white, where it can be harder to identify the black line.

Table 9.3: Statistics on eight Troika proofs; N is the number of components and L_d and L_p give the lines of definitions and proof (excluding comments and whitespace). The last column shows the time VizAssert takes to check each property, and Troika's speedup relative to VizAssert. All times are for Troika with 8 parallel threads.

#	Page	Property	N	L_d	L_p	Initial	Typo fix	VizAssert
1	carshop	button-large	2	6	4	$6s$	$6s$ (1.0×)	$353s$ (64×)
2	puppy	no-text-on-picture	2	7	6	$43s$	$4s$ (11×)	$79s$ (1.8×)
3	tailorshop	three-column	4	9	19	$63s$	$63s$ (1.0×)	$143s$ (2.3×)
4	surf	links-onscreen	6	6	21	$20s$	$20s$ (1.0×)	$42s$ (2.1×)
5	park	links-onscreen	5	6	19	$61s$	$60s$ (1.0×)	$136s$ (2.2×)
6	yoga	links-onscreen	4	6	15	$108s$	$9s$ (12×)	$306s$ (2.8×)
7	yoga	line-width	4	8	8	$80s$	$11s$ (7.5×)	$507s$ (6.3×)
8	yoga	accessible-offscreen	4	8	27	$70s$	$7s$ (10×)	$311s$ (4.4×)

2. The proof author writes a generic specification (a quantified Visual Logic specification for all components that captures the overall logic of the proof without focusing on the implementation of the particular page) and admits it of every component. This generic specification is strengthened until the proof is well-formed.

3. The proof author adds preconditions to each component's specification until the generic specification can be verified using `component-smt`. These preconditions express the details of a page's implementation, and are generally unquantified formulas. The proof author establishes each precondition in the containing component.

In this strategy, the generic specifications ensure that the modular layout proof is well-formed. Then, every precondition is added together with an assertion on another component establishing that precondition, ensuring that the proof as a whole remains well-formed. At every step, it is clear which specification to adjust when a check fails. Troika's ability to admit assertions and mix multiple verification tools supports this workflow.

Comparing the 8 proofs sheds light on these three phases. We found the first phase, page decomposition, easy and quick, taking a few minutes per proof. This phase was especially simple when the theorem focused on a particular part of the page. The second phase, composing generic specifications, usually took fifteen to twenty minutes per proof, and was the most creative portion of the proof. The generic specifications were reusable between proofs of the same theorem for different pages. The third phase, adding preconditions, took the most time: several hours per proof. This time was necessary to understand the source code of an unfamiliar web page and the implicit assumptions that affect its layout, and may be easier for the original page authors. The most common preconditions, governing floating boxes and widths, could be made unnecessary by designing web pages with verification in mind.

9.2.5 Decomposing Pages into Components

The first phase of proof development decomposes a web page into components. For three proofs (1–3 in Table 9.3), we give additional details on this step. Each of these proofs proves a property that applies to a single element on the page. The proofs thus have the opportunity to focus on that element.

In each case, we chose to make the constrained element its own component. For the carshop and puppy proofs (proofs 1 and 2), the rest of the page was the only other component. For carshop, no preconditions are needed, and the proof is two lines long. For puppy, the component of interest requires a minimum width, which is established in the other component; the proof is six lines long. The tailorshop proof (proof 3) is more complex, because the component of interest requires preconditions on the location of floating boxes, which require additional assertions about non-negative margins and floating boxes to prove. These additional assertions are similar to lines 15–20 of the proof in Figure 3.1.

9.2.6 Developing a Generic Specification

The second phase of proof development defines a generic specification for each component. A generic specification contains the core proof strategy, independent of the implementation of a particular web page. This generic specification should imply the theorem (if proven of every component) and capture preconditions that are required of all components. Since a generic specification does not depend on the implementation details of the page in question, we found that it could be reused between proofs of the same property.

Three proofs (4–6 in Table 9.3) provide an illustrative example. All three proofs proved the links-onscreen property and split into different numbers of components. We began by asserting, for each component c, that

$$\forall b, \mathsf{onscreen}(c) \land (\mathsf{is\text{-}link}(b) \lor \mathsf{is\text{-}component}(b)) \implies \mathsf{onscreen}(b) \tag{9.1}$$

For all three proofs, this generic specification was not independently true of most com-

ponents. For example, in the park page (proof 5), if the header were too narrow, the header text would wrap, moving a link off screen. Examining the source code of this page reveals that the header requires a 960 pixel minimum width. This precondition is added in the third phase of proof development.

9.2.7 Adding Component Preconditions

The third phase of proof development adds preconditions to each component until the generic specification can be proven. Since it requires understanding each page's source code, this phase was the most time-consuming.

Determining preconditions for each component requires examining counterexamples and writing preconditions to prevent them. Across the 8 proofs, common patterns emerge. Preconditions for floating boxes are consistently important, appearing in 5 of the 8 proofs, and width preconditions (both minimum and maximum widths) are equally common (also appearing in 5 proofs).

Preconditions for floating boxes required the most assertions to establish. For example, in the proof of links-onscreen for yoga (proof 6, reproduced in Figure 3.1), the footer requires a precondition that no-floats-enter(footer). Establishing this precondition requires four assertions, proving no-floats-enter and no-floats-exit for each component on the page, and also requires proving that each page component has non-negative margins. However, one floating box precondition, in the proof of links-onscreen for park (proof 5), was unusually simple because this page uses the CSS property overflow: hidden, which prevents components' floating boxes from interacting. Web pages designed for verification could use this technique to lower the proof burden.

Because preconditions tend to express the implementation of the page and not the theorem being proved, proofs of different properties on the same page can have similar preconditions. Proofs 6–8 (in Table 9.3) demonstrate both a case of precondition sharing and a case of no sharing. Proofs 6 and 8 (of links-onscreen and accessible-offscreen) share preconditions for footer width (footer.width $\geq$ 200) and floating box non-interference (no-floats-enter(footer)),

which together guarantee that the left-aligned and right-aligned parts of the footer do not overlap or split across multiple lines. On the other hand, proof 7 (of line-width) shared no preconditions with the other two. In this proof, the essential precondition ensures that the text in each component is the correct size; the footer width and floating box preconditions are not useful. This suggests that component preconditions can sometimes be reused between pages, lowering the proof burden, but that proving new theorems sometimes requires the development of new preconditions.

9.2.8 Checking the Proofs

Each proof was checked using Troika on a machine with an i7-4790K CPU, 32GB of memory, and Z3 version 4.5.1. The results are shown in Table 9.3.

The proofs take 8.0–142 seconds, and if multiple cores are available, 5.5–108 seconds (1.1–2.3 × faster).[2] Furthermore, Troika can cache component verifications and reuse them across multiple pages. If the page header, footer, and menus are already cached, such as during incremental development, the proof can be checked even faster, for an average[3] further speedup of 3.2 × over parallel proof checking (column "Typo fix" in Table 9.3). Note especially the large speedup for the yoga page, for which the header is expensive to verify but unlikely to change frequently or across multiple pages.

9.2.9 Comparing Troika and VizAssert

Unlike Troika, VizAssert is not a proof assistant. Though VizAssert does not require writing component specifications, it does not scale to large pages (see Section 9.3), cannot effectively make use of multiple threads, cannot reuse verification effort from one page to another, and supports only a single verification approach. We furthermore found that VizAssert offered little recourse when a theorem took a long time to check, a particular challenge because

[2]These pages all have fewer than 8 components, and often one component is harder to verify than the others, limiting the gain from parallelism.

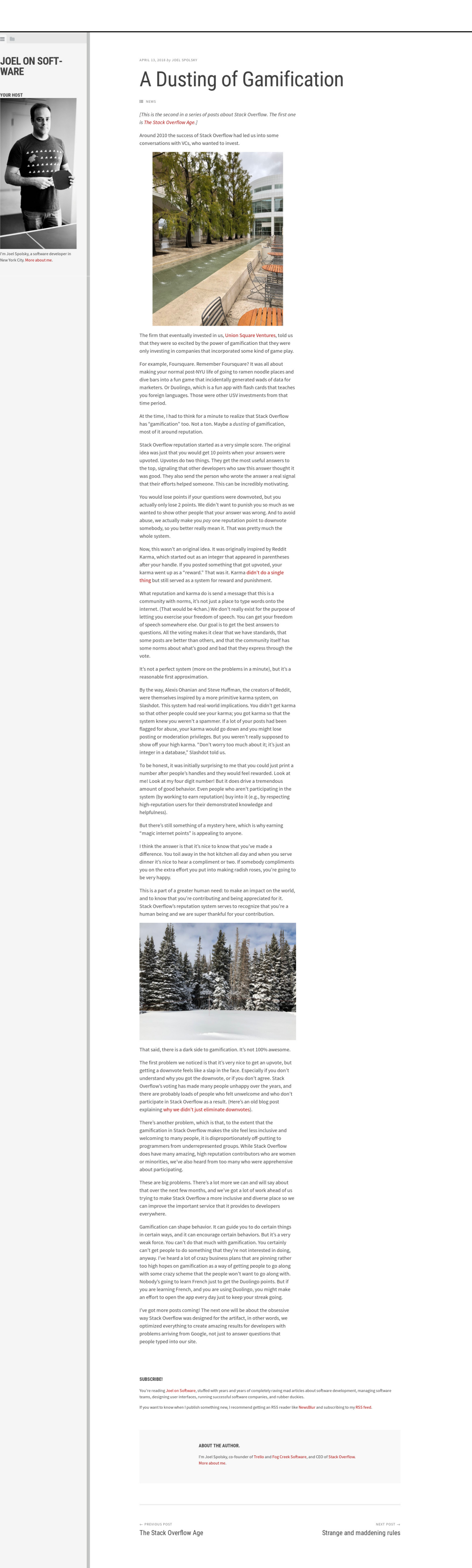
[3]All averages of multipliers use geometric means.

SMT solver variability means that similar formulations of a theorem can have vastly different verification times. Meanwhile, in Troika, pages can be subdivided into more components to speed up verification and make progress on a proof. For large pages, VizAssert's speed is also highly dependent on the particular page being verified because of SMT solver variability. Troika makes many small SMT queries instead of one big one, so is less variable.

Due to VizAssert and Troika's different focus and mode of interaction, comparing the performance of the two tools is difficult. A Troika proof requires choosing a decomposition of the page and writing component specifications, both expensive steps (though proof reuse may lower those costs). However, web pages are edited and changed frequently (to update the content or post user comments, for example). Troika's support for interactive verification and caching allow edits to be made and verified quickly, saving a significant amount of time over the life of a web page. Troika is also on average $2.6\times$ faster than VizAssert when run serially. Furthermore, Troika's architecture allows it to take advantage of caching and multiple threads; Troika is on average $4.2\times$ faster with 8 threads, and $13\times$ faster with caching. The largest advantage for Troika is for the button-large proof for the carshop page (see Table 9.3). button-large concerns a single button on the page; unlike VizAssert, Troika restricts its reasoning to a small component containing the button. Of course, the main goal of Troika is not fast proof checking but the ability to scale to large pages and support interactive web page development styles, as demonstrated in the next section.

9.3 Joel on Software

To demonstrate that Troika scales to large web pages and allows reusing verified components across web pages, we performed a case study with the popular computing blog "Joel on Software" [Spo18]. The verification goal is two properties: all links are scrolling-accessible, and no lines of text are longer than 80 characters. Troika verifies these properties in seconds; VizAssert takes 11–1027 × times longer. Additionally Troika allows reuse *across similar pages* (for other blog posts) and even *across similar sites* (for another blog with a similar theme).



JOEL ON SOFTWARE

YOUR HOST

I'm Joel Spolsky, a software developer in New York City. More about me.

APRIL 13, 2018 *by* JOEL SPOLSKY

A Dusting of Gamification

NEWS

[This is the second in a series of posts about Stack Overflow. The first one is The Stack Overflow Age.]

Around 2010 the success of Stack Overflow had led us into some conversations with VCs, who wanted to invest.

The firm that eventually invested in us, Union Square Ventures, told us that they were so excited by the power of gamification that they were only investing in companies that incorporated some kind of game play.

For example, Foursquare. Remember Foursquare? It was all about making your normal post-NYU life of going to ramen noodle places and dive bars into a fun game that incidentally generated wads of data for marketers. Or Duolingo, which is a fun app with flash cards that teaches you foreign languages. Those were other USV investments from that time period.

At the time, I had to think for a minute to realize that Stack Overflow has "gamification" too. Not a ton. Maybe a *dusting* of gamification, most of it around reputation.

Stack Overflow reputation started as a very simple score. The original idea was just that you would get 10 points when your answers were upvoted. Upvotes do two things. They get the most useful answers to the top, signaling that other developers who saw this answer thought it was good. They also send the person who wrote the answer a real signal that their efforts helped someone. This can be incredibly motivating.

You would lose points if your questions were downvoted, but you actually only lose 2 points. We didn't want to punish you so much as we wanted to show other people that your answer was wrong. And to avoid abuse, we actually make you *pay* one reputation point to downvote somebody, so you better really mean it. That was pretty much the whole system.

Now, this wasn't an original idea. It was originally inspired by Reddit Karma, which started out as an integer that appeared in parentheses after your handle. If you posted something that got upvoted, your karma went up as a "reward." That was it. Karma didn't do a single thing but still served as a system for reward and punishment.

What reputation and karma do is send a message that this is a community with norms, it's not just a place to type words onto the internet. (That would be 4chan.) We don't really exist for the purpose of letting you exercise your freedom of speech. You can get your freedom of speech somewhere else. Our goal is to get the best answers to questions. All the voting makes it clear that we have standards, that some posts are better than others, and that the community itself has some norms about what's good and bad that they express through the vote.

It's not a perfect system (more on the problems in a minute), but it's a reasonable first approximation.

By the way, Alexis Ohanian and Steve Huffman, the creators of Reddit, were themselves inspired by a more primitive karma system, on Slashdot. This system had real-world implications. You didn't get karma so that other people could see your karma; you got karma so that the system knew you weren't a spammer. If a lot of your posts had been flagged for abuse, your karma would go down and you might lose posting or moderation privileges. But you weren't really supposed to show off your high karma. "Don't worry too much about it; it's just an integer in a database," Slashdot told us.

To be honest, it was initially surprising to me that you could just print a number after people's handles and they would feel rewarded. Look at me! Look at my four digit number! But it does drive a tremendous amount of good behavior. Even people who aren't participating in the system (by working to earn reputation) buy into it (e.g., by respecting high-reputation users for their demonstrated knowledge and helpfulness).

But there's still something of a mystery here, which is why earning "magic internet points" is appealing to anyone.

I think the answer is that it's nice to know that you've made a difference. You toil away in the hot kitchen all day and when you serve dinner it's nice to hear a compliment or two. If somebody compliments you on the extra effort you put into making radish roses, you're going to be very happy.

This is a part of a greater human need: to make an impact on the world, and to know that you're contributing and being appreciated for it. Stack Overflow's reputation system serves to recognize that you're a human being and we are super thankful for your contribution.

That said, there is a dark side to gamification. It's not 100% awesome.

The first problem we noticed is that it's very nice to get an upvote, but getting a downvote feels like a slap in the face. Especially if you don't understand why you got the downvote, or if you don't agree. Stack Overflow's voting has made many people unhappy over the years, and there are probably loads of people who felt unwelcome and who don't participate in Stack Overflow as a result. (Here's an old blog post explaining why we didn't just eliminate downvotes).

There's another problem, which is that, to the extent that the gamification in Stack Overflow makes the site feel less inclusive and welcoming to many people, it is disproportionately off-putting to programmers from underrepresented groups. While Stack Overflow does have many amazing, high reputation contributors who are women or minorities, we've also heard from too many who were apprehensive about participating.

These are big problems. There's a lot more we can and will say about that over the next few months, and we've got a lot of work ahead of us trying to make Stack Overflow a more inclusive and diverse place so we can improve the important service that it provides to developers everywhere.

Gamification can shape behavior. It can guide you to do certain things in certain ways, and it can encourage certain behaviors. But it's a very weak force. You can't do that much with gamification. You certainly can't get people to do something that they're not interested in doing, anyway. I've heard a lot of crazy business plans that are pinning rather too high hopes on gamification as a way of getting people to go along with some crazy scheme that the people won't want to go along with. Nobody's going to learn French just to get the Duolingo points. But if you are learning French, and you are using Duolingo, you might make an effort to open the app every day just to keep your streak going.

I've got more posts coming! The next one will be about the obsessive way Stack Overflow was designed for the artifact, in other words, we optimized everything to create amazing results for developers with problems arriving from Google, not just to answer questions that people typed into our site.

SUBSCRIBE!

You're reading Joel on Software, stuffed with years and years of completely raving mad articles about software development, managing software teams, designing user interfaces, running successful software companies, and rubber duckies.

If you want to know when I publish something new, I recommend getting an RSS reader like NewsBlur and subscribing to my RSS feed.

ABOUT THE AUTHOR.

I'm Joel Spolsky, co-founder of Trello and Fog Creek Software, and CEO of Stack Overflow. More about me.

← PREVIOUS POST The Stack Overflow Age

NEXT POST → Strange and maddening rules

PROUDLY POWERED BY WORDPRESS

Figure 9.6: An entry "Joel on Software" blog titled "A Dusting of Gamification", written 13 April 2018, rotated to fit. Note that the page is significantly bigger than those in Figure 9.2.

Table 9.4: Proofs that links are onscreen (links) and that lines are less than 80 characters wide (width) on the "Joel on Software" blog. The joel1 and joel2 pages are posts on the blog and other is a different site using the same theme. In the table, N is the number of components and L_d and L_p give the lines of definitions and proof (excluding comments and whitespace). The last column shows the time VizAssert takes to check each property, and Troika's speedup relative to VizAssert. All times are for Troika with 8 parallel threads.

Page	Prop.	N	L_d	L_p	Initial	Typo fix	VizAssert	
joel1	links	39	6	30	35s	4s (7.9×)	9.9m	(17×)
joel2	links	49	6	30	52s	5s (10×)	9.8m	(11×)
other	links	30	6	30	70s	5s (15×)	33.5m	(29×)
joel1	width	39	8	23	48s	45s (1.1×)	5h 46m	(432×)
joel2	width	49	8	23	68s	45s (1.5×)	19h 26m	(1027×)
other	width	30	8	23	75s	19s (4.0×)	28.2m	(23×)
other*	links	25	6	27	330s	5s (73×)		

9.3.1 Verifying one Blog Post

We chose a post from Joel's blog from April 2018 titled "A Dusting of Gamification" (named joel1 in Table 9.4). The page is 11× larger (by page size) than the FWT pages.

Decomposing the page We decomposed the blog post into components by visual inspection, aiming to create many small components so that each component specification can be verified quickly. We worked hierarchically, first decomposing the page into a sidebar and a content area; then decomposing the sidebar into a title, photo, and description; and then decomposing the content area into components like the article text, an "about the author" blurb, and links to the next and previous post. Of these components, the largest by far was the article text, so we used the ".entry-content > *" selector to subdivide the article text into individual paragraphs and photos. In total, 39 components are created.

Proving links-onscreen We used the generic specification used on the FreeWebsiteTemplates (Equation 9.1 in Section 9.2.6) without change. This assertion was immediately verified for all but two components: the sidebar title and the sidebar.

First, the sidebar title specifies a line height of 40 pixels despite containing 47-pixel-tall text.[4] This meant that the text extended beyond the top of the title box by 3.5 pixels. To prove that the text is onscreen, we added a precondition that the sidebar title is at least 5 pixels below the top of the screen,[5] and asserted this precondition in the sidebar. Second, we added a precondition requiring non-negative margins and non-zero height to several components in the sidebar to prove that these components are not moved off-screen by negative margins.

An additional challenge in verifying these pages is the limited subset of CSS supported by Cassius the sidebar uses the "transform" property and the ":before" and ":after" selectors,

[4] This is may be an error by the web page developer, or it may be a purposeful attempt to achieve tight line spacing.

[5] 5 pixels to allow for cross-platform variation in font metrics.

which are not supported by component-smt. To isolate this issue, we split the sidebar into two components: the sidebar container, which uses the unsupported features, and the sidebar content, which does not. To verify the sidebar container's specification—that the sidebar container's children are onscreen—we manually examined the 9 lines of code in this component. As an additional check, we also ran 10^5 random tests of this component's specification.[6] All other components were verified with the component-smt tool.[7]

Proving line-width Proving that all lines of text are shorter than 80 characters was simpler than proving that all links are onscreen. Unlike some of the pages considered in Section 9.2, in which each element inherits its text size from its parent, the Joel on Software blog post sets an absolute text size on most elements, eliminating the dependence of one component on another. The line width property can thus be proven for the whole page by proving it independently for each component. However, we were unable to do so for the post-footer-widgets component, which holds the "Subscribe!" text. Upon investigating, we determined that this component contained lines of 120 characters, violating the guideline.

To determine whether this is the only component with a line width over 80 characters, we admitted the line width property for the post-footer-widgets component. The full proof then checked, indicating that all other components have shorter line widths.

Checking the Proofs Troika checked the proofs quickly: less than four minutes for each proof when run with a single thread, and about 30 seconds when run with multiple threads (joel1 in Table 9.4). Note that for this page, VizAssert ran 20× and 780× slower, clearly demonstrating the benefits of modular verification. To test Troika's caching abilities, we also constructed variants of the blog post where a single word in the article text is changed, simulating a typo correction. Verifying that links are onscreen takes 4.4 seconds for this

[6]These random tests explored approximately 1% of the total space of browser parameters for this page.

[7]In Table 9.4, the VizAssert times are for a version of the page modified to remove the (small) parts of the page using unsupported CSS features, while the Troika results do not include the time to run random tests.

variant; most of the time is spent checking that the proof is well-formed on the new page. This demonstrates that caching enables mixing verification with incremental development.

9.3.2 Reusing Proofs and Components

Troika enables two kinds of reuse. First, the same component may be present on multiple pages, in which case Troika can cache the component verification, thus reusing the component across multiple pages. Second, the same proof script may apply to multiple similar pages; Troika's ability to define components by CSS selector and to operate on multiple components at a time make this possible even on pages with substantial differences. Two variants of the "Joel on Software" page demonstrate both capabilities.

Reusing components To demonstrate Troika's ability to reuse verified components across web pages, we verified a second post from Joel's blog: "The Stack Overflow Age", also from April 2018. Most of the page content is different, including the title text, publication date, tags, related blog posts, and article text. However, a few components (such as the sidebar and WordPress endorsement) *were* identical between the two pages, and Troika's caching allowed the components to be verified on one page and reused on the other. The tool-specific pruning algorithms for "component-smt" were essential: though the components seemed identical, the pages nonetheless had different CSS style sheets (due to a CSS emoji library) and used different fonts (due to one post using italicized text while the other did not); without pruning, these differences would have prevented caching.

Though the cached components do demonstrate that caching and reuse is possible, they also show its limitations. The 8 reused components (out of 39 total components on the page) are small relative to the article text, so caching them only reduces the overall verification time by 4.2 seconds (12 %). This suggests that cached components are most useful when many similar pages are verified, so that small speedups on each page add up.

Reusing proofs Often, developers reuse components from web libraries, like Bootstrap [OT15], that provide generic themes, layouts, or forms. Ideally, tools like Troika would provide modular layout proofs for these generic library components which can then be reused between multiple sites. Towards this goal, we demonstrate that the proof that verified different posts from Joel's blog can also *verify pages from different sites* that rely on the same third-party library theme.

We investigated the source code of the "Joel on Software" blog and determined that the blog's appearance was based on a lightly modified version of the "Editor" Wordpress theme, a professional theme by the Array Themes design studio [The18]. To check how generalizable our proof is, we attempted to apply it to the theme's demo blog; in other words, to a different site using similar styling. Naturally, all components had different content between the two sites, since even components that do not change across blog posts, like the blog name, still change across different sites. Nonetheless, the specifications we wrote still verified.

The proof applied immediately to the new site, verifying its layout. However, one of the components verified more slowly than we expected: the demo blog had comments enabled, making that component much bigger and causing its verification to take a much longer time (330 seconds for that component; see `other*` in Table 9.4). We made one small modification to improve its efficiency: we moved the comment box to its own component and turned each comment itself into a component. This allowed each comment to be checked in parallel, significantly speeding up verification of the demo blog post. Troika checked the resulting proof in just 70 seconds with parallelism enabled.

Reusing proofs reduces the cost of using Troika. Themes like "Editor" could potentially ship with a Troika script, making it easy for any blog using the theme to prove its accessibility.

Chapter 10
CONCLUSION

This chapter examines limitations, extensions, and other applications of formal reasoning about web page layout.

10.1 CSS Features outside the Cassius Formalization

CSS is large: dozens of standards totaling thousands of printed pages, Cassius supports features to enable verification of a substantial fraction of web pages, but it omits many features as well.

Some features, such as vertical alignment and right-to-left text, would only require engineering time to implement. Over the course of evaluating the Cassius semantics, we extended Cassius multiple times to support new CSS properties: support for the text-align, overflow, white-space, box-sizing, text-indent, and list-style-position properties were all added with minimal changes to the semantics and required only a few hours each to plan, implement, debug, and test each extension. This experience suggests that future extension of Cassius can be done quickly, efficiently, and without large-scale modification of the Cassius semantics. Time permitting, the most important minor features seem to be more complete support for scrolling, the vertical-align property, and :before/:after selectors. However, even with the relative simplicity of adding new features, it is likely neither feasible to completely formalize all of the minor features available in modern browsers, nor keep up with future additions.

Furthermore, some features present significant research challenges in understanding browser implementations, developing finitization reductions, and handling various edge cases. The most important such feature is tabular layout, which occurs infrequently on the web but is substantially different from other layout modes and thus would be a challenge to formalize;

tabular layout is also frequently used in the CSSWG conformance tests, so adding support is important for more complete testing. Text direction (for right-to-left languages) is another significant features: text direction plays a role in many CSS features, and interaction between differently-directed text blocks is a challenging area for formalization. The formalization of text could also be improved, including better support for the `ex` unit and a better model of line breaking. Finally, new CSS features, such as flex-box and even grid layout, are increasingly being used (now that browser support is common), so formalizing these features will become more important with time.

10.2 Extensions to Cassius

Cassius formalizes only the browser layout 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. Cassius-like approaches could also be applied to formalize mobile and desktop GUI frameworks. Alternatively, constraint-based frameworks (such as the iOS Cassowary framework) could be compiled directly to SMT formulas (including specialized handling for priorities). With a formalization of one of these platforms in hand, techniques similar to Visual Logic, VizAssert, and Troika could be used to verify graphical user interfaces.

Cassius 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 generated pages). Future research could extend Cassius to meet this challenge by integrating existing work in symbolic execution of JavaScript [FMSG19] and modeling of DOM manipulations [HLO14]. For example, to verify JavaScript's 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. Temporal operators could be added to visual

logic to express accessibility properties for dynamic code [HBGLB15].

10.3 Debugging and Synthesizing CSS

Visual Logic, VizAssert, and Troika make it possible to automatically verify that web page layouts satisfy accessibility, usability, or design guidelines. However, the core infrastructure enabling these tools—the Cassius formalization of the browser layout algorithm—could be used to develop other novel web page layout tools. To demonstrate this capability, this chapter demonstrates a CSS debugger and synthesizer built on top of Cassius.

10.3.1 Debugging Layouts

To repair a layout bug, such as the product container being too small in Figure 6.1d, the developer first needs to isolate its cause—a fault in the input stylesheet or document. We focus on stylesheet faults. Localizing these faults with an imperative browser involves a manual modify-and-check process, in which individual stylesheet constraints are weakened or strengthened until the original bug disappears from the resulting layout, without introducing new bugs.

With a declarative browser, the fault localization process is automated by *unsatisfiable core extraction*. Using Cassius, we built a prototype debugger that takes as input a concrete stylesheet C, document D, layout B, and the layout parameters $\vec{v}$ in B (e.g., the height of the product container in Figure 6.1d) that violate the desired property $P(B(\vec{v}))$. The debugger converts these inputs into an unsatisfiable layout problem in two steps. First, it translates B into a symbolic layout B' by replacing the value of each $v_i \in \vec{v}$ with a fresh hole h_i. Second, it generates the assertion $P(\vec{h}) \wedge \bigwedge_{h_i \in \vec{h}} h_i = B(v_i)$. By construction, there is no way to complete $\vec{h}$ so that this assertion is satisfied for B', C, and D. Invoking Cassius on such a problem produces an unsatisfiable core—a small subset of the constraints in C, and in the CSS semantics, that is responsible for the violation of $P(\vec{v})$. Our debugger projects this core onto C and pretty-prints it as shown in Figure 10.1.

```
main {background:gray; ~~height~~; ~~float~~}
div { float:left; font-size:144pt;
  width:200px; height:200px; }
```

Figure 10.1: CSS debugger output for the stylesheet, document, and counterexample from Figure 6.1. The localized fault consists of both *set* properties (the float property of <div>), and *unset* properties (the float and height properties of <main>).

10.3.2 Synthesizing CSS from Examples

Having detected and localized the fault in our toy web page (Figure 6.1), we now turn to the problem of repairing the stylesheet so that the product container expands to contain the product icons (Figure 10.1). To automate this task, we used Cassius to implement a *CSS sketching* tool that takes as input a stylesheet with holes and a set of document-layout examples, and fills the holes so that the resulting stylesheet renders each document to the corresponding layout. A stylesheet sketch (Figure 10.2a) contains holes which stand for unknown CSS syntax, while a document-layout example consists of a document and its concrete layout. The synthesizer translates the stylesheet sketch and the document-layout examples into a declarative layout problem using a standard reduction [SLTB+06] of expression holes to numeric holes. If Cassius finds values for the numeric holes, the synthesizer lifts them into a valid completion of the stylesheet sketch. In our running example, the synthesizer repairs the buggy stylesheet from Figure 6.1a to produce the stylesheet shown in Figure 10.2.

10.4 Conclusions

The work in this dissertation demonstrates that the visual appearance of graphical layouts is a valid target for verification, and shows how to extend the tools of automatic program

```
main {background:gray;
  height:?; float:?}
div {float:?;
  font-size:144pt;
  width:200px;
  height:200px; }
```

(a) A stylesheet sketch with expression holes.

```
main {background:gray;
  float: left}
div {float:left;
  font-size:144pt;
  width:200px;
  height:200px; }
```

(b) The completed stylesheet found by a synthesizer.

Figure 10.2: The stylesheet properties identified by a CSS debugger (Figure 10.1) are replaced with expression holes, which the synthesizer fills. Note that the synthesizer ultimately decides not to set the height property.

verification to this new domain. Visual Logic is a compact logic that allows formalizing a range of visual properties of web pages, including design, usability, and accessibility properties. Cassius formalizes the process by which HTML and CSS code is transformed into the layout of a web page. VizAssert combines these two formal statements to allow verifying a Visual Logic specification against the Cassius model of web browsers. And Troika modularizes VizAssert, allowing mixing and matching reusable, verified components when building a web page. The particular techniques developed in the course of this work—the compilation of Visual Logic to SMT, the relational structure of Cassius, the finitization reductions that allow for efficient SMT reasoning, and the definition of independently-true component abstractions—could find analogs and applications in other domains of graphical reasoning, such as mobile applications or visualizations. But more broadly, this work outlines a path toward combining verification with human-centered notions of design, usability, and accessibility, providing a truly end-to-end specification of software.

BIBLIOGRAPHY

[App16] Andrew W. Appel. Modular verification for computer security. *IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016.

[App17a] Apple. Human interface guidelines, 2017.

[App17b] Apple. Ui design do's and don'ts, 2017.

[App17c] Apple Developer. UIKit framework, 2017.

[BBMS99] Greg J. Badros, Alan Borning, Kim Marriott, and Peter J. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology*, UIST'15, pages 73–82, New York, NY, USA, 1999. ACM.

[BBS01] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, December 2001.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

[Big14] Jeffrey P. Bigham. 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, pages 117–122, New York, NY, USA, 2014. ACM.

[Blaa] Black Duck Software, Inc. The chromium (google chrome) open source project on openhub.

[Blab] Black Duck Software, Inc. The mozilla firefox open source project on openhub.

[BLM97] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of the Fifth ACM International Conference on Multimedia*, MULTIMEDIA '97, pages 173–182, New York, NY, USA, 1997. ACM.

[Bro18a] Browserling, 2018.

[Bro18b] Browsershots, 2018.

[Bro18c] Browserstack, 2018.

[Buj15] Zalan Bujtas. `Source/WebCore/platform/LayoutUnit.h`, 2015.

[But10] Matthew Butterick. *Practical Typography*. Matthew Butterick Typography, online only, 2010.

[CCG+03] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30:388–402, 2003.

[Cer11] Lyndon Cerejo. A user-centered approach to web design for mobile devices, 2011.

[CHSA16] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 341–354, New York, NY, USA, 2016. ACM.

[CPO12] S. R. Choudhary, M. R. Prasad, and A. Orso. 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*, pages 171–180, April 2012.

[CSS11] CSSWG. CSS2.1 test suite, 2011.

[CYM10] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1535–1544, New York, NY, USA, 2010. ACM.

[CYM11] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 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, pages 245–256, New York, NY, USA, 2011. ACM.

[Dav12] Burns David. *Selenium 2 Testing Tools: Beginner's Guide*. Packt Publishing, Birmingham, UK, 2012.

[Die00] Diet. Basic act on the formation of an advanced information and telecommunications network society, 2000.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DMS+09] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 429–430. IEEE, 2009.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ekl12] Emil A Eklund. Change fractionallayoutunit denominator to 64 to reduce precision loss when converting to floating point, 2012.

[Eur16] European Commission. 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, 2016.

[FFP+15] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. NSDI'15, Oakland, CA, May 2015. USENIX Association.

[FMSG19] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. Javert 2.0: Compositional symbolic execution for javascript. volume 3, 2019.

[For14] LLC Formstack. Free section 508 compliance checker, 2014.

[Fow17] Amy Fowler. A swing architecture overview, 2017.

[Fre17] Free Website Templates. Free Website Templates, 2017.

[FS06] Barry Feigenbaum and Michael Squillace. Accessibility validation with raven. In *Proceedings of the 2006 International Workshop on Software Quality*, WoSQ '06, pages 27–32, New York, NY, USA, 2006. ACM.

[GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, May 1994.

[HA14] Jonathan Harper and Maneesh Agrawala. Deconstructing and restyling D3 visualizations. UIST '14, 2014.

[HBGLB15] Sylvain Hallé, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. Testing web applications through layout constraints. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8, Graz, Austria, 2015. IEEE, IEEE.

[HBR14] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. Programming by manipulation for layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST'14, pages 231–241, New York, NY, USA, 2014. ACM.

[HLO14] Matthew Hague, Anthony Widjaja Lin, and Luke Ong. Detecting redundant CSS rules in HTML5 applications: A tree-rewriting approach. *CoRR*, 2014.

[HM92] Osamu Hashimoto and Brad A. Myers. Graphical styles for building interfaces by demonstration. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology*, UIST '92, pages 117–124, New York, NY, USA, 1992. ACM.

[HO83] Brent T. Hailpern and Susan S. Owicki. Modular verification of computer communication protocols. *IEEE Transactions on Communications*, 31(1), 1983.

[IC02] Melody Ivory and Aline Chevalier. A study of automated web site evaluation tools. Technical report, University of Washington, Department of Computer Science, 2002.

[JKBD16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269. ACM, 2016.

[JSS+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

[KST+13] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. Webzeitgeist: Design mining the web. CHI'13, 2013.

[KTAK11] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. Bricolage: Example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'11, pages 2197–2206, New York, NY, USA, 2011. ACM.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. PLDI'09, 2009.

[Lei08] K. Rustan M. Leino. This is Boogie 2. Technical report, Microsoft Research, June 2008.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR'10, 2010.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[lev12] leviw@chromium.org. Add fractionallayoutunit type for sub-pixel layout, 2012.

[LKL+13] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. SeeSS: Seeing what I broke – visualizing change impact of cascading style sheets (CSS). In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 353–356, New York, NY, USA, 2013. ACM.

[LMNR15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'15, 2015.

[Lod08] Reed Loden. diff `gfx/src/nsCoord.h`, 2008.

[LSRT15] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 775–782, New York, NY, USA, 2015. ACM.

[MAMH17] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 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, pages 249–260, New York, NY, USA, 2017. ACM.

[MAMH18a] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond. Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 215–226, April 2018.

[MAMH18b] Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G.J. Halfond. Automated repair of mobile friendly problems in web pages. In *International Conference on Software Engineering (ICSE 2018)*, pages 140–150. ACM, 2018.

[MB10] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 711–720, New York, NY, USA, 2010. ACM.

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 260–, Washington, DC, USA, 2003. IEEE Computer Society.

[MCC12] Josip Maras, Jan Carlson, and Ivica Crnkovic. Extracting client-side web application code. In *World Wide Web Conference 2012*. ACM, April 2012.

[MDN17] Mozilla Developer Network. float, May 2017.

[MFT05] Jennifer Mankoff, Holly Fait, and Tu Tran. 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, pages 41–50, New York, NY, USA, 2005. ACM.

[Moz15] Mozilla. `gfx/src/nsCoord.h`, 2015.

[Moz17] Mozilla Developer Network. Mobile accessibility checklist, 2017.

[MP11] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 561–570, May 2011.

[MvC14] Josip Maras, Maja Štula, and Jan Carlson. Firecrow: A tool for web application analysis and reus. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 847–850, New York, NY, USA, 2014. ACM.

[Nat16] National Federation for the Blind. Blindness statistics, November 2016.

[NC15] Tuan A. Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with REMAUI. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'15, pages 248–259, Lincoln, Nebraska, USA, November 2015. IEEE.

[NLWSD14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.

[OAFG98] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. A visual test development environment for GUI systems. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 82–92, New York, NY, USA, 1998. ACM.

[OT15] Mark Otto and Jacob Thornton. Bootstrap: the world's most popular mobile-first and responsive front-end framework, 2015.

[Pam14] Jason Pamental. A more modern scale for web typography, 2014.

[Par17] Parasoft. Web UI testing, 2017.

[Pea17] Pearson. Making e-learning accessible, 2017.

[Pro17] Android Open Source Project. UI overview, 2017.

[RCVO10] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[RGSB00] Murray Rowan, Peter Gregor, David Sloan, and Paul Booth. Evaluating web resources for disability access. In *Proceedings of the Fourth International ACM Conference on Assistive Technologies*, Assets '00, pages 80–84, New York, NY, USA, 2000. ACM.

[RVG15] Azalea Raad, Jules Villard, and Philippa Gardner. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*, volume 9032 of *LNCS*. Springer, 2015.

[SB14] Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.

[SKC+11] Manolis Savva, Nicholas Kong, Arti Chhajta, Li Fei-Fei, Maneesh Agrawala, and Jeffrey Heer. 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, pages 393–402, New York, NY, USA, 2011. ACM.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. ASPLOS XII, 2006.

[SM00] Terry Sullivan and Rebecca Matson. 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, pages 139–144, New York, NY, USA, 2000. ACM.

[Spo18] Joel Spolsky. Joel on software, 2018.

[Sta85] Eugene W. Stark. A proof technique for rely/guarantee properties. In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 369–391, Berlin, Heidelberg, 1985. Springer-Verlag.

[Sut64] Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, pages 6.329–6.346, New York, NY, USA, 1964. ACM.

[T12] Anthony T. Finger-friendly design: ideal mobile touchscreen target sizes, 2012.

[TB13] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. Onward!, 2013.

[TB14] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. PLDI'14, 2014.

[The18] Array Themes, 2018.

[TTA+13] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *POPL'13*, pages 343–356. ACM, 2013.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA'14*, pages 691–707. ACM, 2014.

[UD14] Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 157–176, New York, NY, USA, 2014. ACM.

[US 10] US DOJ. Department of Justice advanced notice of proposed rulemaking, RIN 1190-AA61, 2010.

[US 17a] US DOJ. ADA best practices tool kit for state and local governments, 2017.

[US 17b] US GSA. Section 508: GSA government-wide section 508 accessibility program, 2017.

[vW82] Christopher J. van Wyk. A high-level language for specifying pictures. *ACM Trans. Graph.*, 1(2):163–182, April 1982.

[W3C07] W3C. CSS basic box model, August 2007.

[W3C08] W3C. Web content accessibility guidelines 2.0, 2008.

[W3C11] W3C. Cascading style sheets level 2 revision 1 (CSS 2.1) specification, August 2011.

[W3C16] W3C. WAI-ARIA overview, 2016.

[W3S17] W3Schools. CSS dropdowns, 2017.

[Web17a] Web Platform Tests. web-platform-tests dashboard, wpt/css/css2/floats, 2017.

[Web17b] WebAIM. WAVE web accessibility tool, 2017.

[Wic09] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, New York, New York, USA, 2009.

[Wil05] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[WKM17] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 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, pages 192–202, New York, NY, USA, 2017. ACM.

[WMK15] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. 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)*, pages 709–714, Nov 2015.

[XM07] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1), February 2007.

[ZLSW13] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. The Auckland layout editor: An improved GUI layout specification process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 343–352, New York, NY, USA, 2013. ACM.

[ZM91] Brad Vander Zanden and Brad A. Myers. The Lapidary graphical interface design tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 465–466, New York, NY, USA, 1991. ACM.