# Modular Verification of Web Page Layout

PAVEL PANCHEKHA, University of Utah, USA
MICHAEL D. ERNST, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA
SHOAIB KAMIL, Adobe Research, USA

Automated verification can ensure that a web page satisfies accessibility, usability, and design properties regardless of the end user's device, preferences, and assistive technologies. However, state-of-the-art verification tools for layout properties do not scale to large pages because they rely on whole-page analyses and must reason about the entire page using the complex semantics of the browser layout algorithm.

This paper introduces and formalizes *modular layout proofs*. A modular layout proof splits a monolithic verification problem into smaller verification problems, one for each *component* of a web page. Each *component specification* can use rely/guarantee-style preconditions to make it verifiable independently of the rest of the page and enabling reuse across multiple pages. Modular layout proofs scale verification to pages an order of magnitude larger than those supported by previous approaches.

We prototyped these techniques in a new proof assistant, Troika. In Troika, a proof author partitions a page into components and writes specifications for them. Troika then verifies the specifications, and uses those specifications to verify whole-page properties. Troika also enables the proof author to verify different component specifications with different verification tools, leveraging the strengths of each. In a case study, we use Troika to verify a large web page and demonstrate a speed-up of 13–1469× over existing tools, taking verification time from hours to seconds. We develop a systematic approach to writing Troika proofs and demonstrate it on 8 proofs of properties from prior work to show that modular layout proofs are short, easy to write, and provide benefits over existing tools.

CCS Concepts: • **Software and its engineering → Software maintenance tools**;

Additional Key Words and Phrases: CSS, proofs, modular, layout, verification, SMT

**ACM Reference Format:**
Pavel Panchekha, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2019. Modular Verification of Web Page Layout. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2019), 26 pages.

## 1 INTRODUCTION

Layout bugs can make pages unusable for the millions of users with vision disabilities [National Federation for the Blind 2016] and the billions of users on mobile devices [ITU 2015]. Finding these bugs, or ensuring their absence, is difficult due to the variety of browsers, operating systems, and devices, in addition to user preferences for fonts and colors [Hallé et al. 2015; Mankoff et al. 2005]. Exhaustively testing pages across the infinitely-many possible combinations of these parameters is impossible.

Prior work has begun addressing this challenge by formalizing web layout properties in a logic [Hallé et al. 2015] and developing tools that automatically search for violations of these formal layout properties, either by heuristically analyzing web page source code [Walsh et al. 2017] or using a Satisfiability Modulo Theory (SMT) solver [Panchekha et al. 2018]. Researchers have demonstrated both approaches on small web pages for important layout properties, such

Authors' addresses: Pavel Panchekha, University of Utah, Salt Lake City, UT, USA, pavpan@cs.uw.edu; Michael D. Ernst, University of Washington, Seattle, WA, USA, mernst@cs.uw.edu; Zachary Tatlock, University of Washington, Seattle, WA, USA, ztatlock@cs.uw.edu; Shoaib Kamil, Adobe Research, New York City, NY, USA, kamil@adobe.com.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

as ensuring that web page controls remain accessible even as users increase text size to improve readability.

Current monolithic analyses have several disadvantages. They scale poorly 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 formalizations, soundness, and flexibility, suggests that no single tool is best for verifying every web page.

In other domains, modular verification enables scaling [Appel 2016; Chaki et al. 2003; Dahlweid et al. 2009; Jung et al. 2015] 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.

To scale layout verification to larger pages and sites, we introduce *modular layout proofs*. 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 can constrain such interactions using rely/guarantee-style preconditions [Stark 1985], allowing a component specification to be verified independently of the rest of the page.

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. To capitalize on these benefits, this paper introduces scalable algorithms to check the correctness of modular layout proofs for pages an order of magnitude larger than previous approaches.

We implemented our approach to modular web page verification in a new proof assistant, named Troika. In Troika, the proof author decomposes a web page into components and the layout property into component specifications. Troika checks that the decomposition is valid (using a pure-logic, layout-agnostic algorithm) and verifies the component specifications (using any of multiple verification tools). This allows Troika to scale to larger pages than previous approaches: the layout-conscious verification tools are restricted to small components while whole-page reasoning is scalably layout-agnostic. Troika also allows proof authors to verify different component specifications using different tools (including a whole-page SMT-based verifier, a new specialized verifier for component specifications, a model checker, and a random testing tool), applying each tool where it is most useful and smoothly transitioning from testing to verification.

We performed a case study with Troika on the "Joel on Software" blog [Spolsky 2018], and proved two key accessibility properties; Troika checks the proof 13–1469× faster than existing tools. 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. We also describe a systematic approach to proof construction enabled by Troika, demonstrate it on 8 proofs from prior work, and show that Troika is 1.9–67× faster than existing tools.

In summary, this paper:

(1) Introduces *components*, the units of web page layout modularity, and defines *modular layout proofs*, which prove web page properties from component specifications (Section 4).
(2) Describes an algorithm to verify component specifications independently of the rest of the page (Section 5).
(3) Presents Troika, a proof assistant for web page layout verification, which integrates multiple tools for verifying component specifications (Section 6).
(4) Demonstrates that Troika scales well to pages 11× larger than prior work. Troika verifies the page in 30 seconds, while existing tools take as much as 19 hours, for a speed-up of 13–1469× (Section 7).
(5) Demonstrates a systematic approach to modular layout proof construction with 8 Troika proofs for verification problems from prior work (Section 8).

## 2 BACKGROUND

This paper builds on prior efforts to formalize browser layout and to specify and verify properties of web page layouts.

### 2.1 HTML, CSS, and Browser Layout

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 also consult *browser parameters* such as browser window width and height, end-user font preferences, and browser- and OS-specific values like the size and shape of buttons, input fields, and scroll bars. The *browser layout algorithm* turns HTML, CSS, and browser parameters into a *layout*, which is a tree of *boxes* annotated with position and size information. 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).

*HTML.* An HTML document represents a tree, whose nodes are *elements* or text. Each HTML element has a *tag name* that defines its semantic role (such as "p" for paragraphs), along with other attributes. Some HTML elements, like img, are rendered specially, but most elements' appearance is driven entirely by the CSS properties assigned to them. For example, browsers usually render the h1 element, a top-level heading, using a large bold font. Perhaps surprisingly, this is *not* inherent to the h1 element; it is merely a convention established by page- and browser-defined CSS files.

*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*, written "$\langle$*property*$\rangle$ : $\langle$*value*$\rangle$". 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. This paper focuses on proving properties of static HTML and CSS. Future research could extend it to handle JavaScript by integrating existing work in symbolic execution of JavaScript [Fragoso Santos et al. 2019] and modeling of DOM manipulations [Hague et al. 2014].

## 2.2 Formalizing Browser Rendering

The browser layout algorithm is partially-specified in English-language web standards documents. Several efforts have partially formalized the browser layout algorithm [Meyerovich and Bodik 2010; Panchekha et al. 2018; Panchekha and Torlak 2016]. The algorithm is underspecified, so a page generally has multiple correct layouts. Different formalizations support different CSS properties, and none formalizes CSS in full. The most extensive formalization, part of the VizAssert tool [Panchekha et al. 2018], has been validated against the Firefox browser to ensure that it describes all possible layouts of a page. As part of its evaluation, the authors demonstrated it supports enough of the CSS specification to accurately reason about 62 of a suite of 100 professionally-designed web pages. Anecdotally, VizAssert supports most core CSS features, such as block flow, line layout, floating elements. Among rarer features, VizAssert supports some (shrink-to-fit, positioned layout), but lacks others (:before/:after, table layout, flex-box, right-to-left text). Some missing features, such as right-to-left text or the transform property in Section 7, could likely be straightforwardly formalized. However, some features, such as table layout or flex box, would involve significant effort to formalize, judging by the difficulty of originally implementing these features in existing web browsers. Given that new features are continuously added to CSS, full formalization in the near future is unlikely.

## 2.3 Specifying Layout Properties

Sound formalizations of the browser layout algorithm raise the tantalizing possibility of preventing *layout-based bugs* [Hallé et al. 2015] by verifying *layout properties* that specify which layouts are acceptable and which are not. Verifying a layout property guarantees that all layouts of the page, across a range of layout parameters like browser size, satisfy the property. Several logics for layout properties exist [Hallé et al. 2015].

This paper uses visual logic (Figure 1) [Panchekha et al. 2018]. Logical properties in visual logic are universally quantified over the boxes in the render tree and can use linear arithmetic to express geometric constraints on the size and position of these boxes and their children, siblings, and ancestors. For example, this formula is the running example in this paper:

$$\forall b, b \in \$(\mathsf{a}) \implies b.\mathsf{top} \geq 0 \land b.\mathsf{left} \geq 0$$

It quantifies over all boxes $b$ which match the selector a—in other words, all boxes that are links—and asserts that each link is located below and to the right of the page origin; in other words, it asserts that all links are within the scrollable area of the page. (Note that this property does not restrict boxes from being past the right or bottom edge of the screen, users can see such boxes by scrolling.) Many other usability, accessibility, and interactivity properties have also been formalized in visual logic [Panchekha et al. 2018]. Visual logic can be compiled to the decidable theory of quantifier-free linear real arithmetic, making it appropriate for SMT-based verification.

## 3 OVERVIEW

Troika enables web developers to verify web page layout properties. The developer partitions a web page into components, writing a specification for the behavior 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, without needing to reason about the web page layout algorithm.

$$
\begin{aligned}
\langle \mathit{assertion} \rangle &::= \forall b_1, b_2, \ldots : \langle \mathit{cond} \rangle \\[1ex]
\langle \mathit{cond} \rangle &::= \langle \mathit{cond} \rangle \land \langle \mathit{cond} \rangle \mid \neg \langle \mathit{cond} \rangle \mid \langle \mathit{cond} \rangle \lor \langle \mathit{cond} \rangle \\
&\mid\ \langle \mathit{real} \rangle = \langle \mathit{real} \rangle \mid \langle \mathit{real} \rangle < \langle \mathit{real} \rangle \mid \langle \mathit{real} \rangle > \langle \mathit{real} \rangle \\
&\mid\ \langle \mathit{box} \rangle = \langle \mathit{box} \rangle \mid \langle \mathit{box} \rangle.\mathsf{type} = \langle \mathit{type} \rangle \mid \langle \mathit{box} \rangle.\mathsf{whitespace} \mid \langle \mathit{box} \rangle \in \$(\langle \mathit{selector} \rangle) \\[1ex]
\langle \mathit{real} \rangle &::= \mathbb{R} \mid \langle \mathit{real} \rangle + \langle \mathit{real} \rangle \mid \langle \mathit{real} \rangle - \langle \mathit{real} \rangle \mid \mathbb{R} \times \langle \mathit{real} \rangle \mid \langle \mathit{box} \rangle.\langle \mathit{dir} \rangle[\langle \mathit{edge} \rangle] \\[1ex]
\langle \mathit{box} \rangle &::= b_i \mid \mathsf{root} \mid \mathsf{null} \mid \langle \mathit{box} \rangle.\mathsf{ancestor}(\langle \mathit{cond}^* \rangle) \\
&\mid\ \langle \mathit{box} \rangle.\mathsf{parent} \mid \langle \mathit{box} \rangle.\mathsf{first\text{-}child} \mid \langle \mathit{box} \rangle.\mathsf{last\text{-}child} \mid \langle \mathit{box} \rangle.\mathsf{next} \mid \langle \mathit{box} \rangle.\mathsf{prev}
\end{aligned}
$$

Fig. 1. The grammar of visual logic [Panchekha et al. 2018], an assertion language for formalizing visual layout properties. For brevity, color predicates and keyword definitions are omitted.

This section demonstrates this workflow by stepping through a hypothetical interactive verification session in the Troika proof assistant. The verification session aims to prove that all links are in a scrolling-accessible location on the screen[1] for a yoga studio web page. Figure 2 shows a screenshot of the page and the complete proof.

### 3.1 Writing a Modular Layout Proof

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* $c \in C$, 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 $Q$ when each specification $P_c$ is true of all possible layouts of the component $c$ on the page $p$ and when the conjunction of the specifications $P_c$ imply the whole-page property $Q$, a condition called *well-formedness*:

$$\overbrace{\text{each } P_c}^{\text{specifications}} \quad \text{and} \quad \overbrace{\left(\bigwedge_c P_c\right) \implies Q}^{\text{well-formedness}}$$

**The key innovation of modular layout proofs is that well-formedness must be true layout-agnostically, as a matter of pure logic, independent of the details of web page layout.** This ensures that well-formedness, a whole-page property, can be efficiently checked. It 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. Figure 3 illustrates the overall workflow.

For the yoga studio page, the property $Q$ quantifies over all elements on the page and asserts that elements represented by <a> tags must be to the right and below the page origin. The proof author formally states $Q$ in visual logic (Section 2.3):

```
1   definition scrollable(b) = b.left ≥ 0 ∧ b.top ≥ 0
2   theorem links-scrollable = ∀b, b ∈ $(a) ⟹ scrollable(b)
```

(The line numbers on the code examples in this section match the line numbers in Figure 2.)

With the layout property stated (Step 1 in Figure 3), the proof author can now develop a modular layout proof for it.

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

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

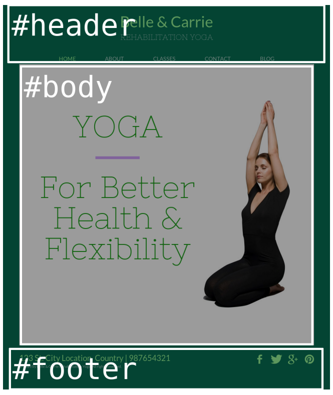

Fig. 2. A complete proof that all links on the yoga page are in a scrolling-accessible location. Section 6.1 describes the semantics of the tactic language. The inset screenshot shows the yoga page and its decomposition into componen

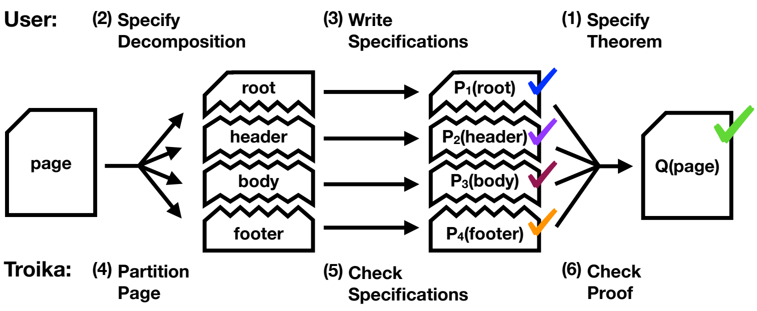

Fig. 3. The Troika workflow. The proof author specifies a theorem $Q$ to prove about a page, describes a decomposition of that page into components (here, the root, header, body, and footer), and writes specifications $P_c$ for each component. Troika uses the decomposition to partition the page into components $c$, verifies each component's specification using various verification tools, and checks that $Q$ follows from the $P_c$.

*Defining Components.* The first part of a modular layout proof is a decomposition of the web page into components (Step 2 in Figure 3). 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 components). For the yoga page, visual inspection suggests subdividing the page into a header, body, and footer (see the decomposition in Figure 2). 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). A Troika specification is expressed using the assert tactic,[2] 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:

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

This assertion ranges over the set $C$ 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. Troika verifies the whole-page property in 0.54 seconds (Step 6 in Figure 3).

## 3.2 Verifying Component Specifications

In a modular layout proof, each component specification may be verified using a different verification tool (Step 5 in Figure 3). 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. This approach is sound but unscalable, so modular proofs commonly employ an alternative approach: 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

[2]Troika provides a language of commands for updating proof state (detailed in Section 6), similar to tactic languages in other proof assistants.

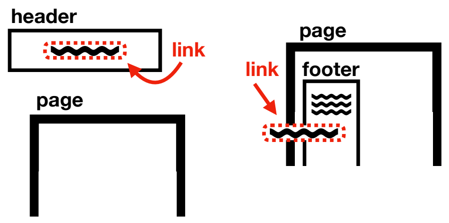

Fig. 4. Two possible problems that the links-scrollable proof guards against using rely/guarantee-style preconditions. (1) If the header is placed off-screen, its links will also be off-screen. (2) If the footer is too narrow, its links will extend beyond its left edge.

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** $c \in C$ **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 4, left). To prevent this counterexample, the proof author would add the precondition scrollable(head):

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

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

~~12~~ **for** root **assert** scrollable(head) **by** admit

Of course, this "admit"ed assertion must later be proven using a sound component verification tool. Thanks to this precondition, the heading can be verified by the component-smt tool, without reasoning about the rest of the page.

The scrollable precondition reflects a common web design convention: 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 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 4, right. The footer thus requires a precondition establishing a minimum width; minimum and maximum widths are common preconditions (Section 8). 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:[3]

13 **for** foot **require** $\mathsf{foot.width} \geq 200$
~~14~~ **for** root **assert** $\mathsf{foot.width} \geq 200$ **by** admit

[3]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.

A proof author can continue to add preconditions to make each component specification independently-true. Figure 2 shows the proof after several more iterations of this process. The proof has floating layout preconditions (described in Section 5.4) 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.

## 4 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 functions or modules. Troika uses a novel definition of a web page component as a subtrees of the web page HTML and a symbolic computed style summarizing its CSS. This section defines web page components and then introduces 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 5 describes how component specifications are checked.

### 4.1 Finding Modularity in Web Pages

Troika uses a partitioning of the HTML tree into subtrees with holes[4] 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 the symbolic computed style that applies to each element, in order to summarize the CSS rules that apply to it.

The *computed style* [W3C 2011] 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 to use that computed value. These guards are used by the solver to define the element's computed style in terms of browser parameters.

Definition 4.1. *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 4.2. *A **decomposition** of a web page into components is a set of components that form a partitioning of that page.*

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

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

Definition 4.3. *A* ***modular layout proof*** *of a property $Q$ for a web page $p$ consists of a decomposition $C$ of $p$ and a* component specification *$P_c$ for each component $c \in C$. A modular layout proof of $Q$ is* well-formed *when $(\bigwedge_c P_c) \implies Q$.*

Both the web page property $Q$ and the component specifications $P_c$ are given in visual logic (see Section 2). To preserve modularity, a component specification quantifies only over the elements within that component. Section 5 describes component specifications further and defines when a component specification is true.

## 4.2 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 5 describes how component specifications are checked to ensure that the overall proof is true.

Unlike verifying component specifications, checking 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, well-formedness can be checked *layout-agnostically*; that is, **no formalization of the browser layout algorithm is used to check the well-formedness of modular layout proofs**. (This contrasts with verifying component specifications, with uses a model of the browser layout algorithm and is thus *layout-conscious*.) 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. It 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 [De Moura and Bjørner 2008] 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.[5] A box consists of an identifier and 28 real-number variables for its size and position.[6] 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 $b$.top[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 "$b = \text{box}_1 \lor b = \text{box}_2 \lor \cdots$", where each $\text{box}_i$ is a known box that matches the selector.

[5] Some elements have multiple boxes (e.g., numbered lists), and some elements have no boxes (e.g., invisible elements).
[6] 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.

The resulting queries are efficiently solvable (see results in Section 8). Because well-formedness is layout-agnostic, the well-formedness of a modular layout proof is usually faster to check than its component specifications.

## 5 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 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.

### 5.1 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 relevant to proving a web page property. For a modular layout proof to be true, the specifications for each of its components must be true:

Definition 5.1. *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 $p$.*

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). Likewise, the height of an element often depends on the heights of its children. These relationships between elements may span multiple 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 scales poorly to larger web pages, with the time to prove a component specification growing with the size of the web page the component is found in.

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 5.2. *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 subtree, 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.

### 5.2 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 5.3 describes component-smt, which checks whether a component specification is independently-true.

Each tool takes as input the web page HTML and CSS,[7] 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 show the Troika user.

*random-test*[$n$] *and model-check.* random-test[$n$] and model-check are based on Cornipickle [Hallé et al. 2015]. 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 set of browser parameters.[8] 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 [Panchekha et al. 2018], and uses an SMT solver to soundly verify component specifications. Although the assertion only refers to a single component of the page, whole-page must still reason about the entire 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 VizAssert.

## 5.3 Verifying Independently-True Component Specifications

The component-smt verification tool is a sound, SMT-based tool for verifying *independently-true* component specifications. Like whole-page, component-smt's implementation is based on VizAssert, which verifies a whole-page layout property $P$ by using an SMT solver to search for renderings of the page that: 1) are valid according to the browser rendering 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 VizAssert 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 rendering 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 possible *effects* of elements outside the component on the layout of the component. By not explicitly representing those elements, component-smt avoids the need for quantifiers. To search for such effects, component-smt ensures that all constraints that relate 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

[7]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.
[8]The model-check only tests integer values of the browser parameters; non-integer values are rare in practice.

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 rendering 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 based on VizAssert.

## 5.4 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.[9] In random-test and model-check, these fields are computed in JavaScript, while whole-page and component-smt use internal state in the browser layout algorithm that tracks this margin collapsing behavior.

*Floating Layout.* The browser layout algorithm moves elements with the "float" property to the left or right of the page; text and other floating boxes wrap around them. Thus, boxes in one component may affect boxes in another. To constrain this interaction, Troika adds the starts-float-free($b, R$) and ends-float-free($b, R$) predicates. Both predicates take a box $b$ and a rectangle[10] $R$. The starts-float-free predicate asserts that floating boxes that precede $b$ do not overlap with $R$,[11] while ends-float-free asserts no overlap for all floating 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 *exclusion zones* [Panchekha et al. 2018] that 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($b$), defined to be starts-float-free($b$, $[-\infty, b.\text{top}, +\infty, +\infty]$), asserts that preceding floating boxes are vertically above $b$, while float-flow-across($b_1, b_2$) 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.

[9]Negative margins are legal in CSS and commonly used for tasks like centering. For these pages, proof authors can define alternative predicates by using $b$.collapsed-margin directly.

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

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

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

$\langle\mathit{tactic}\rangle$ ::= **components** $\langle\mathit{cmp\text{-}name}\rangle$ = \$($\langle\mathit{selector}\rangle$)
  | **for all** $c \in \langle\mathit{cmpset}\rangle$ **assert** $\langle\mathit{assertion}\rangle$ **by** $\langle\mathit{tool}\rangle$
  | **for all** $c \in \langle\mathit{cmpset}\rangle$ **require** $\langle\mathit{assertion}\rangle$

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

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

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

Fig. 5. The core syntax of the Troika tactic language.

## 6 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.

### 6.1 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 which tools to verify them with. Figure 5 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 7.

Within a proof, each "**components**" tactic defines new components, identifying them using CSS selectors. Each element matching 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 7).

The "**assert**" and "**require**" tactics define component specifications: a component with assertions (postconditions) $A_i$ and preconditions $R_j$ has the component specification $\left(\bigwedge_j R_j\right) \implies \left(\bigwedge_i A_i\right)$. 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 $C$ 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.

### 6.2 Implementation

Troika is open-source and freely available online.[12] 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. For checking proof well-formedness, Troika uses the Z3 SMT solver [De Moura and Bjørner 2008]. The whole-page and component-smt verification tools are built by modifying a recent checkout of VizAssert.

To implement the random-test and model-check verification tools, Troika compiles the component specification to JavaScript and uses the Selenium library [David 2012] 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 $b.\mathsf{ancestor}(P)$ expressions to recursive functions, and uses the Range API to determine the size of text. 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.

### 6.3 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 that it does not use. This allows component specifications to be reused across multiple web pages.

Of the four tools provided by the Troika prototype, component-smt makes the most 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.

## 7 CASE STUDY

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" [Spolsky 2018]. 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; existing tools scale poorly to a page of this size, taking 13–1469× 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).

[12] At https://cassius.uwplse.org and https://github.com/uwplse/Cassius.

Table 1. 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 (without caches) 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 | 30$s$ | 4$s$ (8.0×) | 9.9m (20÷) |
| joel2 | links | 49 | 6 | 30 | 44$s$ | 4$s$ (11×) | 9.8m (13÷) |
| other* | links | 30 | 6 | 30 | 66$s$ | 5$s$ (14×) | 33.5m (31÷) |
| joel1 | width | 39 | 8 | 23 | 27$s$ | 20$s$ (1.4×) | 5h 46m (780÷) |
| joel2 | width | 49 | 8 | 23 | 48$s$ | 28$s$ (1.7×) | 19h 26m (1469÷) |
| other* | width | 30 | 8 | 23 | 52$s$ | 11$s$ (4.8×) | 28.2m (33÷) |
| other | links | 25 | 6 | 27 | 304$s$ | 3$s$ (104×) | Same as other* |

## 7.1 Verifying One Blog Post

We chose a post from Joel's blog from April 2018 titled "A Dusting of Gamification" (named joel1 in Table 1). The page is 11× larger (by page size) than the web pages considered in prior work [Panchekha et al. 2018].

*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.

*Proving links-scrollable.* We began by asserting, for each component $c$, that

$$\forall b, \mathsf{scrollable}(c) \land (b \in \$(\mathsf{a}) \lor \mathsf{is\text{-}component}(b)) \implies \mathsf{scrollable}(b) \qquad (1)$$

This assertion was immediately verified for most of the components, but a few required additional preconditions.

First, the sidebar title specifies a line height of 40 pixels despite containing 47-pixel-tall text.[13] This meant that the text extended beyond the top of the title box by 3.5 pixels. To prove that the text is within the scrollable area of the page, we added a precondition that the sidebar title is at least 5 pixels below the top of the screen,[14] 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 the component-smt tool: the sidebar uses the "transform" property and the ":before" and ":after" selectors, 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 within the scrollable area of the page—we manually examined the 9 lines of code

[13]This is may be an error by the web page developer, or it may be a purposeful attempt to achieve tight line spacing.
[14]We chose 5 pixels to allow for cross-platform variation in text rendering.

in this component. As an additional check, we also ran $10^5$ random tests of this component's specification.[15] All other components were verified with the component-smt tool.[16]

*Proving line-width.* Proving that all lines of text are shorter than 80 characters was simpler than proving the prior property. Unlike some of the pages considered in Section 8, 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.

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 1). Note that for this page, existing monolithic verification tools 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. This check is much faster than a proof without caching: links-onscreen, for example, takes 3.8 seconds, with most of the time spent checking that the proof is well-formed on the new page. This demonstrates that caching enables mixing verification with incremental development.

## 7.2 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 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 3.7 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.

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

[16]In Table 1, 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.

*Reusing proofs.* Often, developers reuse components from web libraries, like Bootstrap [Otto and Thornton 2015], 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 proofs for Joel's blog can also *verify pages from a different site* that uses the same third-party library theme.

We investigated the source code of the "Joel on Software" blog and determined that the blog used a lightly modified version of the "Editor" Wordpress theme, a professional theme by the Array Themes design studio [Themes 2018]. 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 (304 seconds for that component). 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 (see other* in Table 1). This allowed each comment to be checked in parallel, significantly speeding up verification of the demo blog post. The modified proofs still apply to the two "Joel on Software" blog posts. Troika checked the resulting proofs in approximately 60 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.

## 8 EVALUATION

This section defines a systematic approach to constructing Troika proofs and demonstrates it on 8 proofs from prior work, comparing similar pages for each phase of proof development. This section then describes the qualitative experience of using Troika and compares it to VizAssert [Panchekha et al. 2018], an existing layout verification tool (Section 8.6). Statistics on these proofs can be found in Table 2. 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 1.9–67×).

### 8.1 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, identifying 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.
(2) The proof author writes a plausible *generic specification* and admits it of every component. This generic specification is strengthened until the proof is well-formed. The resulting generic specification captures the overall logic of the proof without focusing on the implementation of a particular page.
(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

Table 2. 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 (without caches) 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 | 5$s$ | 5$s$ (1.0×) | 351$s$ (67÷) |
| 2 | puppy | no-text-on-picture | 2 | 7 | 6 | 43$s$ | 4$s$ (11×) | 79$s$ (1.9÷) |
| 3 | tailorshop | three-column | 4 | 9 | 19 | 63$s$ | 63$s$ (1.0×) | 141$s$ (2.2÷) |
| 4 | surf | links-scrollable | 6 | 6 | 21 | 20$s$ | 20$s$ (1.0×) | 42$s$ (2.1÷) |
| 5 | park | links-scrollable | 5 | 6 | 19 | 59$s$ | 59$s$ (1.0×) | 134$s$ (2.3÷) |
| 6 | yoga | links-scrollable | 4 | 6 | 15 | 108$s$ | 9$s$ (13×) | 366$s$ (3.4÷) |
| 7 | yoga | line-width | 4 | 8 | 8 | 78$s$ | 11$s$ (7.4×) | 524$s$ (6.7÷) |
| 8 | yoga | accessible-offscreen | 4 | 8 | 27 | 68$s$ | 7$s$ (9.4×) | 247$s$ (3.6÷) |

of a page's implementation. The proof author establishes each precondition in the containing component.

In this strategy, the generic specification ensures 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 learn an unfamiliar web page in order to understand 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.

## 8.2 Decomposing Pages into Components

The first phase of proof development decomposes a web page into components. For three proofs (1–3 in Table 2), 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 four 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. These additional assertions are similar to lines 15–20 of the proof in Figure 2.

## 8.3 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 2) provide an illustrative example. All three proofs proved the links-scrollable property on different pages with different numbers of components. For all three proofs, we used the generic specification from the case study (Equation 1 in Section 7.1) without change.

For all three proofs, the generic specification was not independently true of most components. 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.

## 8.4 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-scrollable for yoga (proof 6, reproduced in Figure 2), the footer requires a precondition that no-floats-enter(foot). 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-scrollable 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 be page- as opposed to theorem-specific, proofs of different properties on the same page can have similar preconditions. Proofs 6–8 (in Table 2) demonstrate both a case of precondition sharing and a case of no sharing. Proofs 6 and 8 (of links-scrollable and accessible-offscreen) share preconditions for footer width (foot.width $\geq$ 200) and floating box non-interference (no-floats-enter(foot)), 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.

## 8.5 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 2.

The proofs take 7.7–139 seconds, and if multiple cores are available, 5.2–108 seconds (1.1–2.4× faster).[17] 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[18] further speedup of 3.1× over parallel proof checking (column "Typo fix" in Table 2). 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.

### 8.6 Comparison to VizAssert

VizAssert is a previous tool for web page verification [Panchekha et al. 2018]. Unlike Troika, VizAssert is not a proof assistant: VizAssert uses an SMT solver to produce a sound guarantee that a web page satisfies a page theorem. Though VizAssert does not require writing component specifications, it scales poorly to large pages (such as the "Joel on Software" pages in Section 7), 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 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 (as in Table 1) because of SMT solver variability. Troika makes many small SMT queries instead of one huge query, 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 2.6× faster than VizAssert when run serially. Furthermore, Troika's architecture allows it to take advantage of caching and multiple threads; Troika is 4.3× faster with 8 threads, and 13× faster with caching. The largest advantage for Troika is for the button-large proof for the carshop page (see Table 2). button-large concerns a single button on the page; unlike VizAssert, Troika restricts its reasoning to a small component containing just 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 Section 7.

## 9 RELATED WORK

### 9.1 Modular Verification & SMT

Modular verification techniques for model checking [Grumberg and Long 1994] pioneered assume-guarantee-style reasoning [Stark 1985] 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 [Chaki et al. 2003] for semi-automatic verification of C programs and Dafny [Leino 2010], 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 [Dahlweid et al. 2009]. One of the major domains for modular

[17]These pages all have fewer than 8 components, and often one component is harder to verify than the others, limiting the gain from parallelism.
[18]All averages of multipliers use geometric means.

verification is for security [Appel 2016], including using finite-state models to modularly verify data transfer protocols [Hailpern and Owicki 1983], and the correctness of compilers [Blazy et al. 2006; Leroy 2006]. Additionally, numerous program logics have been developed to modularly decompose properties of programs, especially in the domain of shared-memory concurrency [Dinsdale-Young et al. 2010; Nanevski et al. 2014; Raad et al. 2015; Svendsen and Birkedal 2014; Turon et al. 2014, 2013], and techniques have also been developed for composing proofs in general [Jung et al. 2016, 2015].

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 [De Moura and Bjørner 2008], 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.

### 9.2 Debugging Web Page Layouts

Many authors have developed 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 [Walsh et al. 2017, 2015]. Other tools attempt to automatically fix errors [Bigham 2014; Mahajan et al. 2018a, 2017, 2018b], using heuristics to detect whether a fix preserves web page correctness. SeeSS aids developers [Liang et al. 2013] by visualizing the impact of edits to CSS stylesheets. Other tools detect parts of web pages that render differently in different browsers [Choudhary et al. 2012; Mesbah and Prasad 2011; Roy Choudhary et al. 2010]. Finally, some tools help developers transfer styles or content between web pages [Maras et al. 2012, 2014]. These heuristic tools are useful, and could be integrated with Troika; however, they do not provide sound guarantees. We are nonetheless excited about these tools as a foundation for building pragmatic tools for quickly debugging proofs of web pages, or as a fall-back when a web page uses CSS features that have not been formalized.

Among practitioners, a large ecosystem of tools exists for designers to test their pages against specific instances of browsers, operating systems, and rendering parameters [Browserling 2018; Browsershots 2018; Browserstack 2018]. Such tools load pages in browsers running in virtual machine instances, returning screenshots to designers so they can manually check against expected renderings. This testing approach is easy to use and widely adopted, but requires manual inspection and, like heuristic tools, does not provide guarantees.

### 9.3 Formal Visual Reasoning

Reasoning about visual layout has a rich history of prior work, including constraint-based systems for specifying and synthesizing layout [Badros et al. 1999; Borning et al. 1997; Hashimoto and Myers 1992; Sutherland 1964; van Wyk 1982; Zanden and Myers 1991]. Similarly, others have explored domain-specific languages [Wilkinson 2005] and visual manipulation [Chugh et al. 2016] for synthesizing graphics programs. Troika is complementary to much of this work; for example, constraint-based approaches for generating components could automatically provide useful preconditions, making their use within Troika easier.

Several authors have produced formalizations of browser layout. Early efforts used attribute grammars to formalize browser layout in order to synthesize parallel layout algorithms [Meyerovich and Bodik 2010]. Cassius [Panchekha and Torlak 2016] formalizes a subset of browser layout in linear real arithmetic in order to synthesize CSS from examples using an SMT solver. Both formalizations are too limited to represent real web pages. Building on Cassius, the VizAssert tool [Panchekha et al. 2018] uses finitization reductions to support a large subset of the CSS standard, including floating layout, which is widely used in modern web pages but is tricky even for experts to reason about. Troika's whole-page and component-smt verifiers build on VizAssert.

Early work on formalizing properties satisfied by web pages, such as Cornipickle [Hallé et al. 2015], focused on detecting layout bugs while testers interacted with a web page. In Cornipickle the properties are given in first order logic with modal operators for representing JavaScript's modifications of the page. VizAssert later adapted Cornipickle's logic to SMT reasoning. Troika uses the same logic to describe layout properties and component specifications.

## 10 CONCLUSION AND FUTURE WORK

Past techniques for verifying web page layout scale poorly to large pages due to their need to perform whole-page verification. We develop a technique for modular verification of web pages, where proof authors decompose large pages into components, write specifications for each component, verify the specifications using a variety of tools, and then prove whole-page properties from the specifications. We describe Troika, a new proof assistant for constructing modular layout proofs, and demonstrate that it scales to large pages, enables parallelism and caching, and allows mixing multiple tools in verification. Compared to prior work, the modular approach is 13–1469× faster and scales to pages an order of magnitude larger than existing tools.

In the future, Troika could be extended to support dynamic content. Most web applications use server-side templates to generate concrete web pages, and those web pages are then modified by JavaScript as the user interacts with them. Troika can verify a particular generated page, or a particular state of the page, but currently not the template or JavaScript code as a whole. However, templates tend to have simple loop structures and JavaScript tends to make localized changes to the page. We hope to explore the possibility of adding some form of inductive reasoning to Troika (across loop iterations in templates and over time as JavaScript changes the page), to allow it to verify dynamic pages in full. We hope that Troika's caching and pruning will provide an additional advantage, since different instantiations of a template or states of a page are often nearly-identical. This inductive reasoning could then be used by a symbolic interpreter for JavaScript to ensure that all possible changes maintain important visual invariants. We see in Troika's future the first steps toward a comprehensive tool for proving visual properties not just for individual pages, but for entire websites and web applications.

## REFERENCES

Andrew W. Appel. 2016. Modular Verification for Computer Security. *IEEE 29th Computer Security Foundations Symposium (CSF)* (2016).

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

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

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006: Int. Symp. on Formal Methods (Lecture Notes in Computer Science)*, Vol. 4085. Springer, 460–475. http://xavierleroy.org/publi/cfront.pdf

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

Browserling. 2018. https://www.browserling.com/

Browsershots. 2018. http://browsershots.org/

Browserstack. 2018. https://www.browserstack.com/screenshots

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2003. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering* 30 (2003), 388–402.

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

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

Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. 2009. VCC: Contract-based modular verification of concurrent C. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 429–430.

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

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

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 504–528. http://dl.acm.org/citation.cfm?id=1883978.1884012

José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *PACMPL* 3, POPL. https://doi.org/10.1145/3290379

Orna Grumberg and David E. Long. 1994. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (May 1994), 843–871. https://doi.org/10.1145/177492.177725

Matthew Hague, Anthony Widjaja Lin, and Luke Ong. 2014. Detecting Redundant CSS Rules in HTML5 Applications: A Tree-Rewriting Approach. *CoRR* (2014). http://arxiv.org/abs/1412.5143

Brent T. Hailpern and Susan S. Owicki. 1983. Modular Verification of Computer Communication Protocols. *IEEE Transactions on Communications* 31, 1 (1983).

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

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

ITU. 2015. ITU releases 2015 ICT figure. http://www.itu.int/net/pressoffice/press_releases/2015/17.aspx

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness *(LPAR'10)*. http://dl.acm.org/citation.cfm?id=1939141.1939161

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54. http://xavierleroy.org/publi/compiler-certif.pdf

Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. 2013. SeeSS: Seeing What I Broke – Visualizing Change Impact of Cascading Style Sheets (CSS). In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 353–356. https://doi.org/10.1145/2501988.2502006

Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G.J. Halfond. 2018a. Automated Repair of Mobile Friendly Problems in Web Pages. In *International Conference on Software Engineering (ICSE 2018)*. ACM, 140–150.

Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated Repair of Layout Cross Browser Issues Using Search-based Techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on*

*Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/3092703.3092726

S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond. 2018b. 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)*. 215–226. https://doi.org/10.1109/ICST.2018.00030

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

Josip Maras, Jan Carlson, and Ivica Crnkovic. 2012. Extracting Client-side Web Application Code. In *World Wide Web Conference 2012*. ACM. http://www.es.mdh.se/publications/2340-

Josip Maras, Maja Štula, and Jan Carlson. 2014. Firecrow: A Tool for Web Application Analysis and Reus. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 847–850. https://doi.org/10.1145/2642937.2648620

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

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

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP (LNCS)*, Vol. 8410. Springer, 290–310.

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

Mark Otto and Jacob Thornton. 2015. Bootstrap: the world's most popular mobile-first and responsive front-end framework. http://getbootstrap.com/

Pavel Panchekha, Adam T. Geller, Michael D Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying That Web Pages Have Accessible Layout *(PLDI'18)*. https://doi.org/10.1145/3192366.3192407

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

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *ESOP (LNCS)*, Vol. 9032. Springer.

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

Joel Spolsky. 2018. Joel on Software. https://joelonsoftware.com

Eugene W. Stark. 1985. A Proof Technique for Rely/Guarantee Properties. In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, Berlin, Heidelberg, 369–391. http://dl.acm.org/citation.cfm?id=646823.706907

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

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS)*, Vol. 8410. Springer, 149–168.

Array Themes. 2018. https://arraythemes.com

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

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

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

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

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

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

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

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