Automated Monitoring of Web User Interfaces

ENNIO VISCONTI, TU Wien, Austria
CHRISTOS TSIGKANOS, University of Athens, Greece
LAURA NENZI, University of Trieste, Italy

Application development for the modern Web involves sophisticated engineering workflows – including user interface (UI) aspects. Such user interfaces comprise Web elements that are typically created with HTML/CSS markup and JavaScript-like languages, yielding Web documents. Their testing entails performing checks to examine visual and structural parts of the resulting UI software against requirements such as usability, accessibility, performance, or, increasingly, compliance with standards. However, current techniques are largely ad-hoc and tailor-made to specific classes of requirements or Web technologies and extensively require human-in-the-loop qualitative evaluations. Web UI evaluation so far has lacked formal foundations, which would provide assurances of compliance with requirements in an automatic manner. To this end, we devise a methodology and accompanying technical framework for web UIs. In our approach, requirements are formally specified in a spatio-temporal logic able to capture both the layout of visual components as well as how they change over time, as a user interacts with them. The technique we advocate is independent of the underlying technologies a Web application may be developed with, as well as the browser and operating system used. To concretely support the specification and evaluation of UI requirements, our framework is grounded on open-source tools for instrumenting, analyzing, and reporting spatio-temporal behaviors in webpages. We demonstrate our approach in practice over Web accessibility standards posing challenges for automated verification.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Logic and verification**; • **Human-centered computing → Web-based interaction**.

ACM Reference Format:
Ennio Visconti, Christos Tsigkanos, and Laura Nenzi. 2025. Automated Monitoring of Web User Interfaces . 1, 1 (October 2025), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Application development for the modern Web[1] is a sophisticated process that involves long engineering pipelines spanning from the specification of business requirements to the worldwide delivery of applications and requires complex continuous integration and deployment workflows. Most of the tasks involved in the process have seen a significant boost in the level of automation and repeatability in recent years. While this is true for many aspects of current development, things are fairly different when considering User Experience (UX) issues. Aspects like usability, accessibility, etc. are becoming increasingly crucial for modern applications, and significant industrial effort is being put

[1]The use of 'web' or 'the web' in this paper refers to World Wide Web sites or apps, and their related technologies as defined by the international standards from the World Wide Web Consortium (W3C), and the Web Hypertext Application Technology Working Group (WHATWG).

Authors' addresses: Ennio Visconti, ennio.visconti@tuwien.ac.at, TU Wien, Vienna, Austria; Christos Tsigkanos, University of Athens, Athens, Greece, christos.tsigkanos@aerospace.uoa.gr; Laura Nenzi, lnenzi@units.it, University of Trieste, Trieste, Italy.

XXXX-XXXX/2025/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

into prioritizing applications that assess them at some level [22]. Validation within a user interface engineering workflow is dominated by User Interface (UI) testing, which concerns mechanisms intended to test aspects of software that a user interacts with. This typically entails inspecting visual elements against requirements - both in terms of functional (such as regulatory conformance), and in terms of non-functional (e.g., performance, timings) ones. Such is the case in contemporary websites, which comprise Web elements created with CSS, JavaScript, and other programming languages, yielding *web documents*. Web UI testing performs tests and checks assertions of these elements to examine visual and structural parts of the software against requirements such as usability, visual design, performance, or increasingly, compliance with standards.

Current web UI testing techniques can be generally categorized as (i) manual-based testing, where graphical screens are checked manually against requirements, (ii) record and replay methods, where automation tools are used to first capture test steps and subsequently execute them on the application under test, and (iii) model-based testing. The latter entails building some representation of the web application, determining inputs and calculating outputs that are used to exercise the application under test, and comparing testing output with what the model expects. Such techniques however are largely ad-hoc and tailor-made to specific classes of requirements and web technologies, or extensively require human-in-the-loop qualitative evaluations. Web UI validation so far has largely lacked formal foundations, which would enable providing (in an automatic manner) assurances on compliance with requirements, something highly desired to check complex e.g., accessibility requirements, so far evaluated manually. To this end, we exploit recent advances of spatio-temporal verification and devise a methodology accompanied by a technical framework for monitoring requirements over web UIs. In our approach, requirements are formally specified in a logic able to capture both aspects of web documents – the layout of visual components as well as how they change over time as a user interacts with them.

The cornerstone of our approach is that UIs can be formalized as a spatio-temporal trajectory $g : \mathcal{L} \times \mathcal{T} \to \mathcal{P}(C) \times E$, where locations of $\mathcal{L}$ correspond to locations of the graphical device (e.g., pixels of the screen), time points of $\mathcal{T}$ correspond to the graphical refresh frames, $\mathcal{P}(C)$ is a power set of graphical UI components (e.g. buttons, images, input fields, etc.) and E are interaction events that a user may induce. The advantages of a formal perspective in such a specification come from the fact that it can be defined and monitored regardless of the specific physical device or browser being used, allowing automated support for a wide set of use cases in interaction simulation and testing, which today are heavily platform-specific.

Our contributions target automated monitoring of web UIs, lie within a novel application of formal methods in the contemporary Web engineering workflow, and are as follows:

- We propose a methodology for the monitoring of web UIs, accompanied by a technical framework integrated with current Web technologies;
- We show how Web documents can give rise to formal models, whereupon a spatio-temporal logic can be used to express general requirements;
- We illustrate WebMonitor– an end-to-end technical framework, where, given a webpage target, and a specification written in a convenient DSL, automated procedures carry out analysis[2];
- We demonstrate automated monitoring of requirements sourced from the widely-applicable Web Accessibility Standards (WCAG2.1 [15]), and investigate verification performance over different screen sizes and browser engines.

The approach we advocate is independent from the underlying technologies a web application is developed with, as well as from the browser/operating system in use (e.g. Google Chrome, Microsoft

[2] WebMonitor has been previously presented as a tool demo at [57], and can be found at github.com/ennioVisco/webmonitor.

Edge, Mozilla Firefox, both used interactively and headless, are supported out of the box) – a stark difference from existing approaches. Moreover, it can be easily extended to non-web applications (e.g. videogames' UIs or desktop applications), provided proper instrumentation is in place. Our implementation is open-source software, and can be found in the accompanying material along with an experiment reproduction kit.

The rest of the paper is structured as follows. Sec. 2 provides an overview of our approach and introduces a motivating example, used throughout the paper. Section 3 presents spatial and temporal models of web UIs, while Sec. 4 discusses logic-based reasoning on such models. Sec. 5 presents the instrumentation of a technical framework supporting web UI monitoring, including its key architectural features. Sec. 6 showcases how our approach can be used in practice, Sec. 7 discusses the evaluation of our approach, Sec. 8 summarizes related work and Sec. 9 concludes the paper.

2 OVERVIEW: WEB UI MONITORING

Software application development for the modern Web culminates in an artifact – the resulting web UI, which is the product deployed and finally delivered to users. To this end, we seek mechanisms intended to test the aspects of web application software with which a user interacts. Those reflect increasingly important non-functional requirements of contemporary web applications such as usability, accessibility, or compliance with standards. In contrast to other efforts in the domain, we pursue an automated way of assessing whether web UI designs conform to stated requirements. Requirements can be quite complex, as they may predicate on spatial arrangement and characteristics of visual elements as well as sequences of user interactions.

Fig. 1 illustrates a birds-eye view of the domain and proposed approach for automated monitoring of Web user interfaces. The development process revolves around the creation of a UI, which should satisfy certain requirements assumed to be elicited from stakeholders. Those are formalized into properties, while from the user interface, a spatio-temporal UI model is derived. Such a model captures the spatial arrangement and characteristics of visual elements as well as interaction sequences inherent in the desired User Experience (UX), and is intended to be *analyzable* in an automatic manner. Subsequently, the model and properties are given as input to an *oracle*, which produces a *verdict* – an evaluation result of whether the model satisfies the stated properties. In case of violations, a *visual counterexample* is returned, which is shown to the developer. The last step is crucial, and highlights the framework's place within the development cycle: the developer may revise the application and invoke the cycle again. The whole process is cast within a continuous integration or continuous delivery (CI/CD) workflow, where spatio-temporal model extraction, analysis, and counterexample generation are performed in an automated manner.

Running example. Consider a cookie consent notice – a banner informing the user about the cookies stored on the browser to track a website's usage. Cookie consent notices are enforced by an increasing number of regulations, most notably GDPR [56] in the European Union and CCPA [14] in California, USA. These normative acts prescribe precise levels of disruptions in users' perception of the page and also establish binding sanctions that can even reach 4% of a company's annual revenue for non-compliance with the regulation [31]. Such a cookie popup must be easily dismissible by the user. The design of cookie consent notices has to satisfy certain functional and non-functional requirements, some of which may be imposed by regulations [15, 56] directly, others that derive from common design practices [49], and some that may come from internal design guidelines. We distill some characteristic ones:

ER1 The popup should be visible to the user. As a visual component, the entirety of the popup should be within the window that the user perceives as the interface so that all the relevant

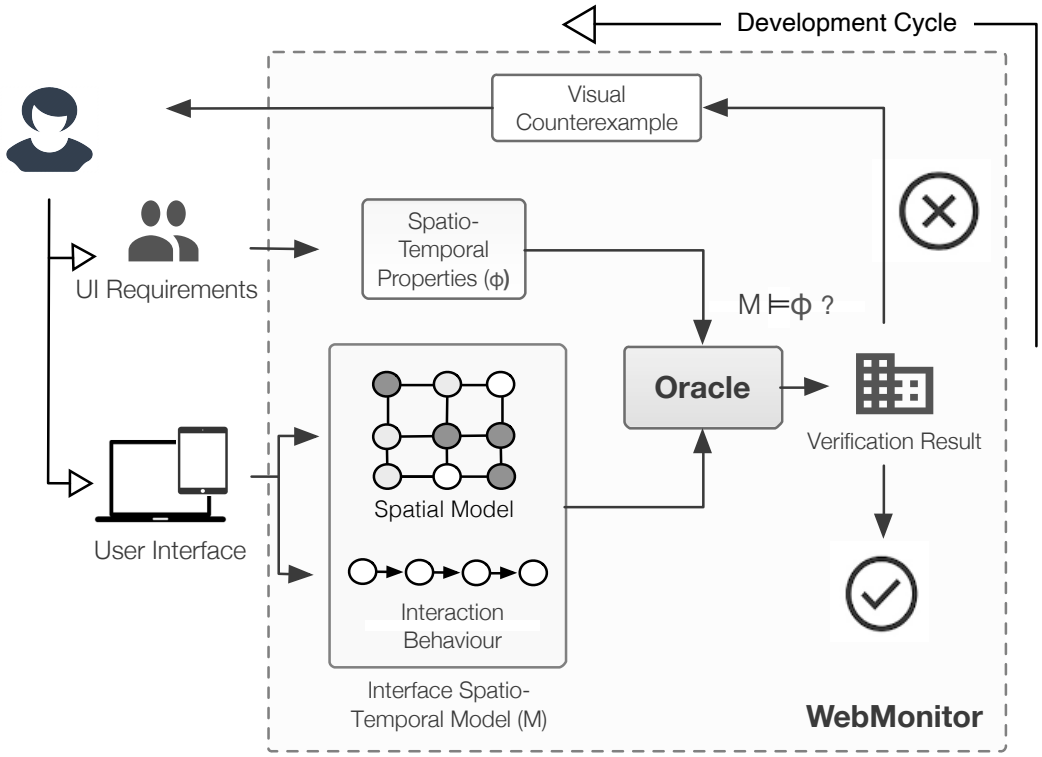

Fig. 1. Automated monitoring of web user interfaces.

information is available from the beginning of the navigation session (e.g. reasons for usage, link to policies, acceptance, denial, and cookie settings buttons).

ER2 The popup should be *dismissible*: if the user clicks on some relevant button (e.g., to accept or deny cookie permissions), the popup should disappear.

ER3 The popup should be visible (ref. ER1) to the user *within two seconds* after the page has been loaded; it should *remain visible* until the user explicitly dismisses it (ref. ER2).

Observe that the example describes a characteristic case that involves both *spatial* and *temporal* aspects of the UI; visibility of the popup (ER1) can be inspected upon a snapshot of the UI, while dismissibility (ER2) involves checking if the popup disappears over a sequence of user actions. The last requirement (ER3) exemplifies a complex behaviour since it requires that both ER1 and ER2 hold but also denotes timing constraints for the UI design of this component.

Supporting the evaluation of such requirements upon a web page design in an automated way, calls for a dedicated methodology and technical framework. Those should be able to i) capture requirements and UI design in an unambiguous way amenable to automated analysis, ii) instrument appropriately external components (such as browser APIs), and iii) contextualize analysis and interpretation of results within the typical development cycle.

3 SPATIO-TEMPORAL MODELS OF WEB UIS

A (graphical) user interface (UI) is a software system for human-computer interaction, involving functional components and behavioral events placed in areas of a graphical space (i.e., the screen),

evolving over time as the interaction with the user unfolds. In the web's context, UIs are challenged by variations in devices that will effectively run them. However, such web UIs are also grounded on rules and established technologies that abstract the specificity of the operating system and browser vendors of the device that is running them, allowing for platform-agnostic analysis methodologies such as the one advocated in this paper. From the perspective of an external observer, graphical UIs can be seen as multi-valued *trajectories* over a 2-D graphical space. In the following, we elaborate on such a conception in detail. Specifically, we consider the graphical space as a *spatial model* $\mathcal{S}$ and the multi-valued trajectories as a function $g : \mathcal{L} \times \mathcal{T} \rightarrow \mathcal{P}(C) \times E$, where $\mathcal{L} \times \mathcal{T}$ represents the spatial and temporal dimensions respectively, $\mathcal{P}(\cdot)$ is the power set of C, representing a set of functional components (e.g. buttons, images, input fields, etc.), and E denotes a set of behavioural events (text-area focused, data retrieved, button clicked, etc). Below, we formalize each of these elements.

3.1 Spatial Model

Graphical user interfaces assume the presence of a display for interacting with the user – this is the case in all platforms, including the Web. Therefore, an intuitive interpretation of the 2-D graphical space corresponds to a set of (x, y) coordinates that identify each pixel of the pixel-grid that composes the physical display of the targeted device. More formally, we devise a spatial model $\mathcal{S} = \langle \mathcal{L}, \mathbf{W} \rangle$, where:

- $\mathcal{L}$ is a set containing all the logical locations the objects can occupy i.e., the pixels, identified by the coordinate pairs $\ell = (x, y)$, with $x, y \in \mathbb{Z}$ i.e., $\mathcal{L} = \{(x_{min}, y_{min}), ..., (x_{max}, y_{max})\}$;
- $\mathbf{W} \subseteq \mathcal{L} \times \mathcal{L}$ represents a proximity relation, such that an arc of the graph $(\ell_1, \ell_2) \in \mathbf{W}$ if the two locations are adjacent (i.e. given $\ell_1 = (x_1, y_1)$ and $\ell_2 = (x_2, y_2)$, either $x_2 = x_1 \pm 1$ or $y_2 = y_1 \pm 1$).

This formalization allows to describe the pixel grid as a spatial model where distances can be efficiently computed by the L1/Manhattan definition (i.e. the distance between ℓ_1 and ℓ_2 on the grid is expressed by $|x_1 - x_2| + |y_1 - y_2|$). Observe that the spatial model does not necessarily need to map exactly to the grid of physical pixels, and further representation optimizations can be defined depending on the granularity of the specification under analysis.

In most cases, these areas correspond physically to fully-connected rectangular pixel grids. Recent technological advancements are exploring different, more complex, layouts (e.g., circular screens for smartwatches, dual-screen foldable devices, modular screens); the model presented can also accommodate such layouts.

Recall the running example; requirements `ER1` – `ER3` entail checking the behavior including any conditions that move a given UI component in or out of the user's focus. More generally in the context of the web, three conceptually different areas of the screen should be represented:

(1) the **document**, which is the logical region over which the web page is defined throughout its lifecycle; it contains all possible objects of the page, some of which may not be visible or accessible by the user (e.g. an object at position x: -1000px, y: -1000px is not visible but it is still computed and part of the page).
(2) the **layout viewport**, which represents the reachable area of the page. This is usually bigger than the targeted display but can be accessed by users' interactions (e.g., swiping or scrolling).
(3) the **visual viewport**, which is the region of the page that is displayed to the user at a given moment. It usually corresponds to the inner frame of the browser window, or to the whole display (when in full-screen mode).

Fig. 2 illustrates these areas for a hypothetical web page. A requirement like `ER1` would translate to the popup being in the visual viewport (3) until the user explicitly dismisses it.

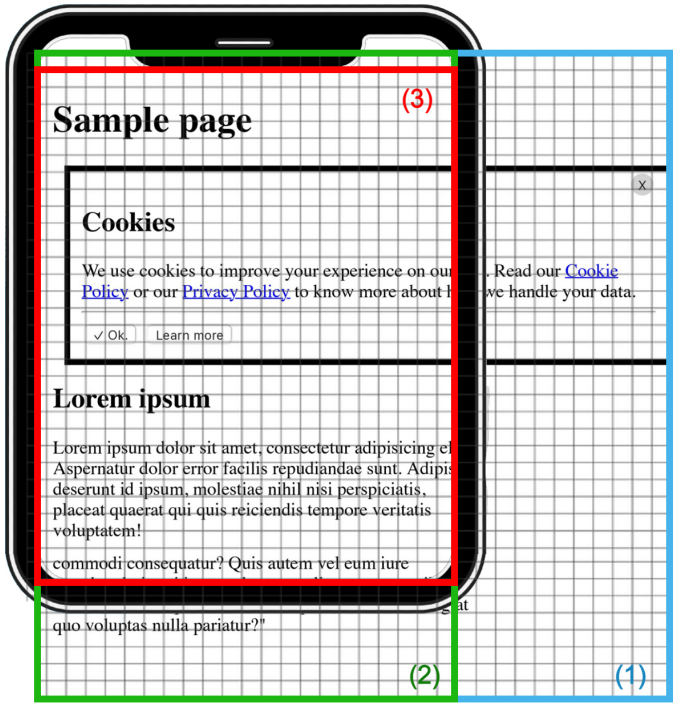

Fig. 2. Example of a web page with a cookie consent notice. Colored boxes with solid borders represent the different conceptual areas of the page. The cyan box (1) represents the overall document, the green box (2) shows the layout viewport and the red box (3) denotes the visual viewport.

3.2 Components and Events

First-class objects of a graphical user interface are components and events deriving from the user's interaction with them. Thanks to the standardization effort that has characterized the Web since its conception, components are defined by developers utilizing the HTML standard, as published by the Web Hypertext Application Technology Working Group (WHATWG).

```
< div class = "cookieInfo" > We use cookies to... < /div >
```

The previous line shows a typical implementation of the outer box of the cookie popup of Figure 2. The `div` HTML element represents a generic container. In addition, the `class` attribute `cookieInfo` identifies it within a specific category defined by the developer, which may be associated with styling instructions for the browser to display it appropriately. Such tags may be nested, forming a tree hierarchy.

Standard web APIs provide a wide range of events fired by the browser engine when the user interacts with the page[3]. Without loss of generality, we consider for presentation purposes the primary browser events in E: *click*, *focus*, *scroll*, and *load*. Those are fired respectively when the user clicks an element, selects an element, scrolls the page, and when the page loading is completed. The browsers first interpret the page's components, then attach styling rules, set behavior watchers, and render

[3]Only Document Object Model (DOM) events are considered as the reference API. Future iterations may also target the Web Audio API and `AnimationFrame` information to allow specifications on audio and video elements.

the final result to the user's screen. We consider the set of components C to coincide with the set of HTML elements present on a given web page. Typical elements are e.g., `div` that represents a generic container and `a` that represents an anchor with a hyperlink (either to another section of the page or to a different web page). In addition to them, we add to this set a pseudo-tag that we call `screen` that acts as a container for all the elements that are present in the visual viewport. Given the set of HTML elements C, a set of events fired by user's interaction E, the trajectory of a page in position $(x, y) \in \mathcal{L}$ and at time $t \in \mathcal{T}$ is of the kind: $g(x, y, t) = (\{\texttt{div}, \texttt{a}, \texttt{screen}\}, click)$, with $\texttt{div}, \texttt{a}, \texttt{screen} \in C$ and $click \in E$. Note that the event is the same in all the locations and there can only be a single event at each time-step. Our approach includes all HTML elements, properties, and events; we employ a subset in the paper to simplify the presentation.

3.3 Temporal Dimension

Any realistic analysis of a graphical user interface must take into account the dynamics of the interface as the user interacts with it. This dynamics can be represented in various ways , resulting in different notions of the temporal dimension $\mathcal{T}$ (a typical alternative approach is one based on abstractions [17]). From the viewpoint of the user, changes that happen visually at a frequency higher than the refresh rate of the screen (usually 60 Hz), are not perceived – most events typically occur at much lower frequencies since users perceive a response as being instantaneous when happening within 0.1s [41] and as such we consider interactions that receive a response within 1s. However, one does not need to perceive events as occurring with respect to a refresh rate when analyzing web UIs, in fact, the actual flow of events recorded by the browser's engine can be considered instead: while user interactions can happen at a high refresh rate (e.g., the user moves the mouse), only a few of them trigger effects on the UI (e.g., when the mouse points over a clickable link). Such events fire in the browser's event loop, a standard HTML concept that guarantees synchronous recording of user events [59], and can be accessed programmatically, employing a browser engine.

Whether we denote the time $\mathcal{T}$ as the ordered set of events $= t_0, t_1, ..., t_n$ that are recorded by the browsers' event loop, or by the continuous-time that is perceived by the user, we can assume user interface changes as described by a piecewise-constant time signal, meaning that a UI g can be effectively analyzed at the time points t_i of interest for the given specification. For example, a prototypical temporal trace describing the evolution of the web page of Fig. 2 would be characterized by a sequence that starts at t_0, when the page is loaded and the spatial model is derived. After a *click* event, a new time-point is generated representing the new state of the page. As such, the temporal model expresses evolution based on user events. We will refer to graphical UIs and web UIs interchangeably, as they are equivalent within our approach. The whole system is then described by the spatial model and the trajectory. In our simple example, $g(x, y, t_0) = (\{\texttt{div}, \texttt{a}, \texttt{screen}\}, load)$, $g(x, y, t_1) = (\{\texttt{div}, \texttt{a}, \texttt{screen}\}, click)$ would define the spatio-temporal trajectory for two time-steps.

4 REASONING ON UI TRAJECTORIES

To evaluate properties on the spatio-temporal trajectories of UIs, a specification language to express meaningful properties predicating in both space (the UI) and time (the user behavior) is required. The language we advocate is based on *Spatio-Temporal Reach and Escape Logic* (STREL) [38]. We opt for describing the relevant features of the logic over the running example and the requirements described in Section 2. For a complete formal treatment and semantics, the interested reader is referred to [38]. A STREL formula conforms to the following grammar[4]:

$$\varphi := \langle\ \texttt{atomic}\ \rangle \mid \langle\ \texttt{boolean}\ \rangle \mid \langle\ \texttt{temporal}\ \rangle \mid \langle\ \texttt{spatial}\ \rangle \tag{1}$$

[4]We simplify temporal and spatial operators by using t and d to mean $[0, t]$ and $[0, d]$ respectively, in contrast with the original logic using intervals to define these bounds, since we found them superfluous for our use cases.

Atomic propositions. The basic building blocks of STREL specifications (Formula 2) for web UIs are atomic propositions (or *atoms*), predicating on attributes of the page elements.

$$\langle \texttt{atomic} \rangle := p \circ c \mid b \mid id \sim f(p) \tag{2}$$

These can be either (i) inequalities or equalities with respect to constants of interest or to some other attribute value of the page elements i.e., ~~$\mu := p \circ c$~~ with $\circ \in \{<, >, =, \leq, \geq\}$, or (ii) they can be directly Boolean values in that location i.e., ~~$\mu := b$~~. For example, requirements ER1 – ER3 are centered on the visibility of the cookie popup which is identified by the developers-defined class `cookieInfo`. Note that in principle there could be more than one popup (e.g., a website could have a small one at the bottom of the page and a big one at the center), yet the properties would still be valid.

$$\mu_{visible} := \texttt{.cookieInfo\$visibility} = \text{`visible'}. \tag{3}$$

Formula 3 encodes that the visibility property of all the elements of class `cookieInfo` has the value `visible`. To retrieve a specific element of the page in an atomic property, we adopt the standard W3C selector notation [21], followed by the special character '$' to denote the exact styling property being analyzed. When appropriate for the specific formula, in line with [21], we will mark with `:` the specific HTML state of the element (e.g. `button : active` to denote a property that is *active* – has just been clicked). An example of Boolean atomic proposition is the `screen` property that is true only in areas of the spatial model that are currently shown on the user's screen – this property implies specification of the pixels that belong to the visual viewport.

In addition to standard STREL atomic propositions, we introduce in Formula 2 a special set of atomic propositions , i.e., $id \sim f(p)$, where '$\sim$' is the bind comparator. Such comparator captures the first value satisfying the property selector 'p', it applies the function f to this value and stores it to the identifier id. Subsequent references to the identifier id will be compared to the previously stored value and evaluated to true only when the values are the same. Note that the binding value is captured once throughout execution, and therefore serves as a parameter that is constant within the trace. With such atomic propositions, values of interest useful in web UIs can be tracked and compared, e.g.:

$$\begin{aligned} &\texttt{p\$color} \sim \texttt{specialColor} \\ &\texttt{section\$background-color} \sim \texttt{specialColor}. \end{aligned} \tag{4}$$

The first to be satisfied among these two atomic propositions will set the value of the '`specialColor`' identifier (to e.g., red), while the subsequent will be marked as not satisfying it when having a different value (being e.g., black).

Boolean Operators. The basic operators one can use to express a specification are the ones of classical logic (Formula 5) that represent respectively *negation*, *conjunction*, *disjunction*, as well as the *implication* of some subformulae φ.

$$\langle \texttt{boolean} \rangle := \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \tag{5}$$

For example, in the context of a cookie popup, the requirement ER1 would be formalized as follows:

$$\phi_{\texttt{ER1}} := \mu_{visible} \wedge \texttt{screen}. \tag{6}$$

Formula 6 states the basic condition that a popup must be visible *and* that at the same time or location the `screen` property must hold. Therefore ER1 will be satisfied only in the areas of the spatial model that are on the screen and have a visible popup.

The requirement ER2 would instead be formalized as

$$\begin{aligned} \mu_{hidden} &:= \texttt{.cookieInfo\$visibility} = \text{`hidden'} \\ \phi_{\texttt{ER2}} &:= (\texttt{button.close : active} \rightarrow \mu_{hidden}). \end{aligned} \tag{7}$$

Formula 7 expresses the behaviour related to clicking the closing button. In fact, when the `button` element with class `close` is clicked (i.e., it becomes active), then ($\rightarrow$) it must be invisible ('`hidden`').

Temporal operators. To predicate about behaviour, temporal operators expressed in Formula 8

$$\langle \texttt{temporal} \rangle := \mathrm{F}_t\ \varphi \mid \mathrm{G}_t\ \varphi \tag{8}$$

are used to express the fact that a subformula φ is satisfied *for some* (F_t) - respectively *for all* (G_t) - next time points t, as in usual temporal logics. To illustrate that, consider ER3, which involves that both ER1 and ER2 should hold within two seconds:

$$\phi_{\texttt{ER3}} := \mathrm{F}_{2s}\ (\phi_{ER1} \wedge \phi_{ER2}). \tag{9}$$

Spatial operators. To predicate about the relative distance between elements of the space, spatial operators of Formula 10

$$\langle \texttt{spatial} \rangle := \diamondsuit_d \varphi \mid \boxdot_d\ \varphi \tag{10}$$

are used to express the fact that a subformula φ is satisfied *for some* ($\diamondsuit_d$) - respectively *for all* ($\boxdot_d$) - locations within a distance of d. Spatial operators enable a concise yet general way to express how the user may access alternative elements; for example, the following formula describes that all accept buttons must be at least 10 pixels distant from cancel buttons:

$$\texttt{button.cancel} \rightarrow (\boxdot_{10px} \neg \texttt{button.accept}) \tag{11}$$

The logic comes with efficient monitoring procedures [38]. Given a STREL formula ϕ, and the pair $\langle S, g \rangle$, where $\mathcal{S}$ is a spatial model, and $g : \mathcal{L} \times \mathcal{T} \rightarrow \mathcal{P}(C) \times E$ a spatio-temporal trace, the monitor computes a Boolean function, $\beta_{\langle S,g \rangle} : \mathcal{L} \times \mathcal{T} \rightarrow \mathbb{B}$ that returns the Boolean satisfaction[5] of the property in each location at each time. This means that the Boolean function is equal to 1 in location ℓ, at time t, i.e. $\beta_{\langle S,g \rangle}(\ell, t) = 1$, iff $\langle S, g(\ell, t) \rangle$, in location ℓ at time t, satisfies the property ϕ, and 0 otherwise. The entire $\langle S, g \rangle$ satisfies ϕ, i.e. $\langle S, g \rangle \models \phi$ iff $\beta_{S,g} = 1$ for all locations and time steps. When this is not the case, WEBMONITOR maps back to a screenshot of the page the failing areas at the specific time steps when the properties fail, providing a visual counter-example.

Consider again the running example, and suppose we want to apply the monitoring algorithms to formula ϕ_{ER1}. Imagine a typical interaction flow on a device like the one of Fig. 2 at a $800x600$ resolution, where, after the page has fully loaded, the user reads the cookie popup and grants the related permissions. Such a sequence would generate a trace structured as such: $g(x, y, t_0) = (\{\texttt{screen}, \texttt{.cookieInfo}\}, \mathit{load})$ in the area where the cookie popup is visible, say between $20 \leq x \leq 800$, and $100 \leq y \leq 500$; $g(x, y, t_0) = (\{\texttt{screen}\}, \mathit{load})$ for $x \leq 800, y \leq 600$, and $g(x, y, t_0)$ maps either to $\emptyset$ or to $\{\texttt{.cookieInfo}\}$ in all other (x, y) exceeding the screen. After the button is clicked, the event generates a new snapshot. Therefore, $g(x, y, t_1) = (\{\texttt{screen}\}, \mathit{click})$ for any $x \leq 800, y \leq 600$. Consequently, for ϕ_{ER1} the monitoring output will be $\beta(x, y, t_0) = 1$ only when g maps to $(\{\texttt{screen}, \texttt{.cookieInfo}\}, \mathit{load})$, while $\beta(x, y, t_1)$ will always be 0 (since $\mu_{visible}$ does not hold anymore).

5 INSTRUMENTING WEB UI MONITORING

Having discussed the underlying theoretical foundations, appropriate instrumentation is required to be in place to realize an end-to-end solution. This refers to automating the interfacing with browsers, generating spatio-temporal trajectories, and interpreting analysis output in a meaningful visual way. Fig. 3 shows a dataflow diagram of WEBMONITOR. The overall process starts with a *Web Source* as an evaluation target, which denotes a URL as well as auxiliary parameters required such as the browser and screen size. Subsequently, the process can be considered as taking place in three stages:

- *Tracking*: The first stage of the WEBMONITOR workflow is responsible for the execution and collection of the webpage data. The `Web Source` descriptor is used to launch a browser session that will be used to fetch elements and events as described by the atomic propositions found in the `Spec`. Possible interactions with the page must be injected at this stage, either by

[5] We use 1 and 0 in place of `true`, `false` to represent the Boolean interpretation, in line with [38], as the use of numbers is more in line with the algebraic characterization of the semantics

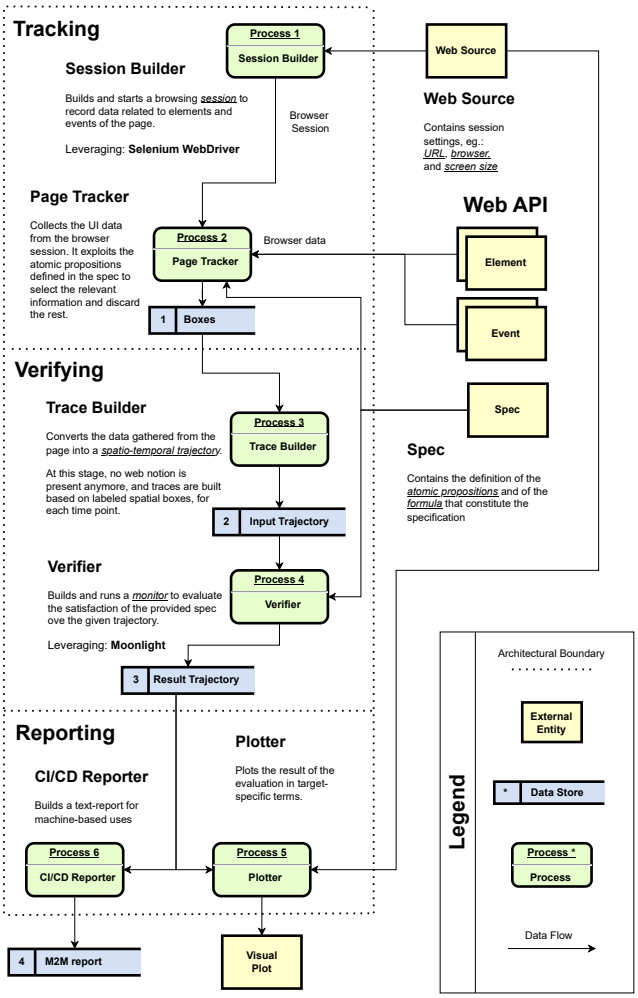

Fig. 3. Dataflow diagram including primary architectural boundaries of WEBMONITOR.

direct interaction with the page, or by using any related automation tool[6]. The concrete output of this stage is a set of bounding boxes[7] for each time-point; each box is labeled corresponding to the identifier defined in the atomic proposition and has sizes corresponding to the computed position by the browser. Selenium Webdriver[8] APIs are leveraged for the interaction with web browsers, by using instructions that work interchangeably across browsers, and therefore limiting to the W3C WebDriver standard [53].

- *Verifying*: The second stage is responsible for the actual analysis of the specification with respect to the data collected at the previous stage. It starts via the `Trace Builder` that

[6]For example automa.site or zennolab.com.
[7]DOMRect interfaces, drafts.fxtf.org/geometry/#DOMRect.
[8]selenium.dev

analyzes the collected data and the session settings, to generate a spatio-temporal multi-valued trajectory of the kind described in Section 3. Then, the `Verifier` parses the provided `Spec` to generate a monitor corresponding to the specification, and launches the evaluation on the spatio-temporal trajectory. At the end of the verification process, a new artifact is generated, which is the binary map of the satisfaction values of the specification, for every location and time point of the trajectory. The specification classes and the monitoring facilities are provided by the Moonlight[9] library for monitoring STREL formulae.

- *Reporting*: The last stage is responsible for delivering the results of the evaluation. This may be manifested in two ways, faithful to contemporary processes within web application development. The first is intended to integrate analysis within a typical developer workflow, where a `Plotter` facility generates a figure highlighting the areas of the target screen where the specification is violated. This represents the *counterexample*, as in classical model checking, although it is extracted from the observed trace, corresponding to the exact event that led to that incorrect state. The developer may revise the design accordingly and restart the verification process. The second form of reporting functionality targets continuous integration and continuous delivery (CI/CD) pipelines. To this aim, we employ a set of configuration scripts that turn WEBMONITOR into a no-dependency tool, which can be adopted in any environment, manifested as Github Actions that automatically run it on a given source and specification, upon commits pushed. In this case, the `Reporter` facility generates a machine-readable result of the evaluation, intended to be consumed by the appropriate pipeline stage (e.g., among integration or other testing hooks), perhaps as a means for deciding whether or not to deploy the application in production.

To use the advocated framework in practice, a developer follows four distinct steps:

(1) Initialization; a target web page is specified, along with parameters concerning browser and screen size.
(2) Specification; requirements that the design should fulfill are specified via the logic outlined in Sec. 4.
(3) Analysis; verification facilities are invoked.
(4) Reporting; in case of violation, the visual counterexample is inspected.

We note that our implementation is open-source software[10] which can be found in accompanying material along with an experiment reproduction kit. The reader interested in a more hands-on perspective of WEBMONITOR's usage can refer to the tool demonstration available at [57] or a video tutorial[11].

6 USAGE IN PRACTICE

We see WEBMONITOR as a foundational building block for a new generation of monitoring and testing tools. As such, in this section we both present how the approach can be employed in practice, both in terms of the DSL we developed to operate it, and also in terms of how it can be exercised to support more sophisticated development scenarios.

6.1 WEBMONITOR's DSL

WEBMONITOR has been designed to be accompanied from the beginning by a powerful domain specific language (DSL) that simplifies its adoption in several contexts. The DSL is developed via the

[9] Moonlight STREL library github.com/MoonLightSuite/MoonLight.
[10] A snapshot version of the WEBMONITOR artifact with experiments replication package at the time of writing can be found at: https://figshare.com/s/f0d63ddf60370098aac6
[11] Video demonstration of WebMonitor: youtu.be/hqVw0JU3k9c.

Kotlin compiler, which allows us to provide a full-featured development experience out-of-the-box when WEBMONITOR scripts are opened in IntelliJ IDEA (syntax highlighting, code completion, documentation, error reporting, etc.) – from which the subsequent screenshots are taken – or any other IDE supporting Kotlin's DSL features. A WEBMONITOR's DSL file is typically identified by the extension `.webmonitor.kts`, which usually contains a few lines that identify the *preamble* as in Fig. 4. The first line is a standard shell instruction, which is used to instruct the running environment about the compiler to call (therefore, it is only required to run WEBMONITOR's DSL scripts as standalone shell programs). The second line informs the compiler about the version of the WEBMONITOR library to load (this can also be omitted when WEBMONITOR is incorporated in an ANT, Maven or Gradle project). Lastly, the third line loads the DSL, which allows the specification to be written and parsed correctly.

```
1  #!/usr/bin/env kotlin
2  @file:DependsOn("com.enniovisco:webmonitor:1.3.1")
3  import com.enniovisco.dsl.*
```

Fig. 4. WEBMONITOR's DSL's preamble.

Following the preamble, the developer needs to define the monitoring scenario, which is described in a `monitor {...}` directive, and is composed of the *Web Source* and *Spec* descriptors introduced in Sec. 5. The *Web Source* is identified by the `webSource {...}` directive, which contains the session information of the scenario, as shown in Fig. 5. For most of the fields presented in Fig. 5, sensible defaults are provided (e.g. the predefined browser's window width and height), with the exceptions of `maxSessionDuration` and `targetUrl`, which are the two pieces of information always required by the developer, identifying, respectively, the maximal duration of the scenario, and the URL of the web page to evaluate. The default browser is Google Chrome in normal mode, although in several instances developers prefer the screen-detached version (denoted by the `*_HEADLESS` suffix).

```
5  monitor {
6      webSource {
7          screenWidth = 393 // px
8          screenHeight = 851 // px
9          browser = Browser.CHROME_HEADLESS // Or FIREFOX, EDGE, CHROME...
10         wait = 0 // ms
11         maxSessionDuration = 5_000 // ms
12         targetUrl = "https://enniovisco.github.io/webmonitor/sample.html"
13     }
14     // ...
15 }
```

Fig. 5. WEBMONITOR's DSL's *Web Source* specification.

Lastly, the *Spec* to analyze is identified by the `spec {...}` directive, which is decomposed in the following required instructions:

atoms(...) which contains a comma-separated collection of atoms that can be used in the specification. This directive informs WEBMONITOR about the elements of the document and/or their related properties to track (see Fig. 6 for some examples of atoms). To facilitate the definition, the following keywords are available to the programmer:

- `select {...}`: this keyword constructs an atom based on the CSS query selector that is passed between the brackets. It is the minimal construct required to define an atom.
- `read`: given the previous CSS query selector, this optional keyword can be used to access the value of a specific CSS property (e.g. background, visibility, ecc.) for the corresponding elements of the page.
- `equalTo` (or other comparators): each time a CSS property is read, a comparator must be provided, so that the atom can be properly evaluated to true or false. Several comparators are already provided by WEBMONITOR (e.g. `equalTo`, `greaterThan`, `lessEqualThan`, etc.). Alternatively, developers can define their own custom comparator that is appropriate to the specific value being retrieved or the kind of properties they want to write.

record(...) contains a comma-separated collection of conditions that should trigger a new retrieval of information on the page (see Fig. 6). They are identified by the `after{...}` keyword, containing a string corresponding to the identifier of a standard web event to track (e.g. click, touch, close, etc.).

formula = ... is the last instruction of a WEBMONITOR specification and denotes the formula to monitor. It is often desirable to decompose the specification in subformulae (see Fig. 7). The atoms are referenced in order of appearance in the atoms collection (see Fig. 6), the unary operators wrap the subformulae in '`()`', and the binary operators are infixed.

```
5   monitor {
6       // ...
7       spec {
8           atoms(
9               select { ".cookieInfo" }    // [0]
10                      read "visibility"
11                      equalTo "visible",
12              select { ".subtitle" },     // [1]
13              select { "button#close" },  // [2]
14          )
15
16          record(
17              after { "click" },
18              after { "touch" }
19          )
20          // ...
21      }
22  }
```

Fig. 6. WEBMONITOR's DSL's *Spec* specification (first part): atoms definition and trigger events.

```
5    monitor {
6        // ...
7        spec {
8            // ...
9
10           // helper formulae
11           val popupIsVisible = atoms[0]
12           val subtitle = atoms[1]
13           val closeButton = atoms[2]
14           val er1 = closeButton implies screen
15           val er2 = not(popupIsVisible and subtitle)
16
17           // Final formula
18           formula = everywhere(er1) and er2
19       }
20   }
```

Fig. 7. WEBMONITOR's DSL's *Spec* specification (second part): helper and final formulae.

6.2 WEBMONITOR: exemplar workflows

A developer can encompass WEBMONITOR in a wide range of scenarios to support specifying, monitoring and testing. In the rest of this section, we present how it can be incorporated in traditional development flow, while Sec. 7 will present a more complex use case for monitoring accessibility.

Regression Testing Workflow. We present here a workflow where WEBMONITOR is used as a standalone regression testing facility, a typical scenario spanning common aspects of the development process of an application.

(1) A user interface bug is found and reported by the Quality Assurance (QA) team or an end-user (e.g., the button to close a popup is cut out from the screen).
(2) The developer based on details of the bug report writes a minimal *Web Source* (e.g. like the one shown in the previous section) to load the page and reproduce the conditions of the reported bug. Optionally, the developer might integrate WEBMONITOR in more complex page-interaction tools, like Selenium[10], depending on how complex are the conditions to reproduce the bug.
(3) The developer then proceeds to write a minimal *Specification* of the correct behavior (recall Listing 1, which shows a possible way to formalise ϕ_{ER1}).
(4) Finally, the developer sets a continuous delivery pipeline (an example for Github Actions is provided on the official repository[12]), and at each commit to the shared repository, WEBMONITOR is run against the provided web source and specifications, reporting a failure when the specification is not satisfied.

Listing 1. Specification definition in WEBMONITOR DSL for a regression testing scenario.

```
spec {
  atoms(
        select { ".cookieInfo" } read "visibility" equals "visible"
```

[12]Relevant links and tutorials can be accessed at: github.com/ennioVisco/webmonitor

```
    )

    val isVisible = atoms[0]
    formula = isVisible and screen // Final formula
}
```

BDD Specification workflow. In this workflow WEBMONITOR is used as the tool to enable Behaviour-Driven Development (BDD [42]) executable specifications of a new feature:

- The business experts/product owner come to the development team requesting a new feature: supporting payments in the software product.
- At the start of the development sprint, the development team meets and discusses an informal specification of the feature.
- In addition to the functional specifications, they decide that the following specification makes sense for this feature: "in no case the button to execute the money transfer should be close to other buttons" – to avoid inadvertently completing the transaction.
- Before developing the feature, one the developer takes the responsibility in formalizing the specification, and writes the code shown in Listing 2, deciding that *not close* means that they must be at least 20px apart.
- The new WEBMONITOR specification is added to the list of acceptance tests of the software, and from now on, developers implementing the technical details of this feature will see the test fail if this requirement is overlooked.

Listing 2. Specification definition in WEBMONITOR DSL for a BDD development scenario.

```
spec {
  atoms(
        select { "input[type=submit]" },
        select { "button, input[type=button]" }
  )
  formula = atoms[0] implies (everywhere(not(atoms[1])) within 20)
}
```

6.3 User considerations

Formal specification and reasoning, despite their well-known effectiveness, are often not preferred by developers [39], which are the intended users of our approach. We assume this to be the case also and especially within web development. To use WEBMONITOR in practice, one needs to be familiar with i) web technologies and ii) the STREL logic as outlined in Sec. 4. Observe that specifications within our approach take place over CSS Selectors [21], a notation that the intended users in the web context would be highly familiar with. This is a deliberate design choice in order to enable faster learning; with CSS selectors developers can write specifications with the novel component being solely the logic part. We further note the avoidance of introducing a new independent syntax in WEBMONITOR, relying on the Kotlin DSL – the default language for e.g., Android apps or the Gradle building tool.

We do acknowledge that formal requirements reasoning may pose difficulties – a concern quite often raised within the formal methods community regarding adoption. Research efforts have long pursued to bridge this gap from different angles, from specification patterns easing formulae writing [36], to interactive requirements reasoning tools (notably e.g., FRET [20]) or via refining specifications via high-level modeling techniques [28]. A promising direction is also outlined by the efforts leveraging natural language processing (see [24] for a work translating a fragment of the logic

we are adopting), or more recent Large Language Models' aid in formal specifications (like in the technical reports [1, 60]). We consider future integration of such formal requirements elicitation and reasoning techniques as of having high potential in practice for WEBMONITOR.

7 EVALUATION

Utilizing the WEBMONITOR framework, in the following we demonstrate its usage in practice over requirements sourced from widely-applicable web accessibility standards. Specifically, we first describe how such regulatory requirements can be formalized. Subsequently, we showcase their verification against an arbitrary webpage, illustrating the framework's versatility in an end-to-end manner. Finally, we present our experimental setup, illustrate the qualitative results obtained, and conclude with a discussion.

7.1 WCAG2.1 Analysis Scenarios

Among the kinds of requirements a designer can express via our approach, international web accessibility standards provide a natural source for general and widely applicable requirements often overlooked because of the challenges that their analysis may require. The current practice for verifying these requirements is via human-in-the-loop evaluations [26, 50] that reproduce as closely as possible realistic interaction patterns of the final users. While these approaches are undoubtedly beneficial, automation could cover many tasks without relying on specific hardware configurations.

The EU Web accessibility directive [13] (and Action 64 of the EU Digital Agenda), and similarly the US Section 508 amendment [37] target the Web Content Accessibility Guidelines (WCAG 2.1) [15], a set of UI guidelines belonging in four principles – (*Perceivable*, *Operable*, *Understandable*, *Robust*). Often quite generic, they express desired abstract functionality and visual or structural layout of web documents. We select ones amenable to automated visual UI reasoning; thus, we ignore other aspects, such as audio-related ones or captions in images, and ones that can be statically checked for compliance (e.g., by directly inspecting the HTML source code, without the interpretation that a browser implies). We treat WCAG as regulatory requirements and note that the source can be *any* webpage – for demonstration purposes, we select the New York Times homepage[13], over which we simulate several interaction patterns. We note that the proposed formalizations are indicative, as there may be other equivalent ones.

Perceivable. This WCAG principle intends to assess that the information and user interface components must be presentable to users in ways they can perceive. We focus in particular on the following "reflow" [15] requirement.

(*Reflow*) Content can be presented without loss of information or functionality, and without requiring scrolling in two dimensions for: (i) vertical scrolling content at a width equivalent to 320 CSS pixels, and (ii) horizontal scrolling content at a height equivalent to 256 CSS pixels.

To formalize this requirement, we consider the primary *text-content* HTML tags, i.e. paragraphs (p), and seek to assess that their size is strictly below the indicated threshold for at least one of the two dimensions, as captured in Formula 12.

$$(\texttt{p\$height} < 320px) \vee (\texttt{p\$width} < 256px) \tag{12}$$

The perceivable principle of WCAG is the most extensive and covers many aspects of UI perception; another important aspect is ensuring that tooltips and hoverable elements are working correctly, as in the following "hover" [15] requirement.

(*Hover*) If pointer hover can trigger additional content, then the pointer should be able to be moved over the additional content without the additional content disappearing.

[13] New York Times, https://nytimes.com

Formula 13 captures a partial formalization of this more complex requirement. We encode this property specifically for the source website considered; in such context, it is reasonable to require that when an item (`li`) of the navigation (`nav`) list is hovered (`:hover`) by a pointer, a `.secondaryNav` element must appear within a half-second ($G_{0.5s}$), somewhere around it ($\diamondsuit_{1px}$ - in at most $1px$ of distance).

$$(\texttt{nav li:hover} \rightarrow G_{0.5s} \diamondsuit_{1px} \texttt{.secondaryNav}) \tag{13}$$

Lastly, a key requirement – often posing difficulty for automated analysis – is one requiring a contrast ratio between elements that allows the user to easily read the content:

(*Contrast*) The visual presentation of text and images of text has a contrast ratio of at least 7:1.

A possible formalization of this requirement is the one of Formula 14. In this case, we assign the text color (`color`) of all the article titles (`h3`) within a container (`div`) to the identifier `titlesColor`, and we require that when the titles have that color, the respective parents (`div : has(h3)`) have a background color (`background – color`) that corresponds to a contrast-ratio of `titlesColor`, γ being a helper function comparing the relative contrast-ratio, according to the specification.

$$\begin{aligned}&(\texttt{div h3\$color} \sim \texttt{titlesColor}) \wedge \\ &(\texttt{div:has(h3)\$background-color} \sim \gamma(\texttt{titlesColor}))\end{aligned} \tag{14}$$

Operable. This principle refers to the ability of UI components and navigation to be operated easily by the users. We focus in particular on the "three flashes requirement" [15].

(*Flashes*) Web pages should not contain elements that flash more than three times in any one-second period.

To specify this requirement precisely, we consider elements changing display state rapidly, which in the case of the target website is the real-time stock-exchange information block. Formula 15 encodes this requirement, stating that when some stocks' highlight (ν_{stocks}) is present on the screen, it must not be the case that it subsequently disappears, and that, after it, a (possibly different) highlight appears again.

$$\begin{aligned}&\nu_{\text{stocks}} := \texttt{.masthead-bar-one-widgets div} \\ &(\texttt{screen} \wedge \nu_{\text{stocks}}) \rightarrow (\neg(F_{0.5s}(\neg\nu_{\text{stocks}} \wedge F_{0.5s}(\nu_{\text{stocks}}))))\end{aligned} \tag{15}$$

Understandable. By understandable, the WCAG prescribes that a typical user must be able to quickly deduce the relevant information and operations of the UI. We consider the following "focus" WCAG requirement.

(*Focus*) When any component receives focus, it does not initiate a change of context.

We illustrate the formalization of this requirement as it applies to typical UI elements: pop-up ads. Formula 16 formalizes this requirement. Once popup ads appear (`.welcomeAdLayout`), users might spend some time reading them, perhaps using the mouse pointer to guide their focus (`hover`). For as long as this is the case (G_∞), we expect the popup to stay visible.

$$\begin{aligned}&\nu_{\text{adPresent}} := \texttt{.welcomeAdLayout} \\ &\nu_{\text{adHovered}} := \texttt{.welcomeAdLayout:hover} \\ &\nu_{\text{adHovered}} \rightarrow G_\infty(\nu_{\text{adPresent}})\end{aligned} \tag{16}$$

The presented Formulae 12-16 have been written with the goal of obtaining an evaluation trace, allowing to see the exact time-point in which the property is not satisfied. When the intended usage is to get a single truth value that summarizes the total satisfaction of the formulae in all time-points (respectively pixels of the spatial model), a G_∞ (respectively $\boxdot_\infty$) should be put at the beginning of all of them.

7.2 Experimental Results

To utilize our approach in practice, the designer specifies the desired properties (Sec. 4, Sec. 6, Sec. 7.1), and establishes the scope of the analysis by identifying the web source and the settings of the browser session (browser engine, screen resolution, and maximum duration of the session, Sec. 5). The workflow of Fig. 3 is then automatically performed. Experiments were conducted on a (multi-threaded) Apple M1 Pro with 16GB RAM, using the Google Chrome browser engine.

Table 1. Experiment results on WCAG2.1 fragment analysis.

	Spatial Size (px)	**# affected elements**	**# of Events**	**Memory Peak** (MB)	**Execution Time** (min)
Reflow	568x320	2	16	2470	1:24
Hover	568x320	1	3	1070	22:25
Contrast	568x320	266	3	600	2:31
Flashes	568x320	0	34	3800	1:52
Focus	568x320	1	4	1500	0:43
Reflow	800x600	2	15	2450	1:21
Hover	800x600	2	2	1300	20:55
Contrast	800x600	375	3	927	3:19
Flashes	800x600	1	30	3600	1:39
Focus	800x600	1	4	1470	0:43

Table 1 summarizes the evaluation of the different WCAG requirements of Sec. 7.1, over different screen resolutions (column "Spatial Size"). The spatial model targets two specific resolutions, 393x851 which is the resolution of the Pixel 5a smartphone, and 800x600 which used to be the most common resolution among old monitors. The choice to target relatively old resolution configurations is intentional within web UI testing – they typically represent edge cases where modern designs fail, and for that reason are subject to dedicated testing efforts. The proposed specification has been tested with manual interaction when appropriate (e.g., moving the mouse over appropriate elements) and with *timeout* events between $300ms$ and $15s$ when considered useful for the specific property. Further details about the tested trace are provided in the replication package. All experiments have been conducted for a browsing session of the same exact duration of $23s$, included in the final time of Table 1. Different choices can be made taking into consideration specific user interactions with the webpage. The central columns report the maximum number of page elements that have been affected by a requirement during the evaluation of the trace (# affected elements) and the number of events recorded throughout that trace (# of Events). The rightmost columns report the maximum resident memory during the evaluation and the total execution time. These include all three stages of Fig. 3 – from session building and interaction with browser engines to verification and counterexample generation, although we note that the Verifying stage of Fig. 3 – building and running a monitor to evaluate the satisfaction of a provided property over the given spatio-temporal trajectory – was responsible for almost the entirety of the execution time beyond the browsing session. Conversely, the most memory-intensive operation is the counterexample generation, where results of the evaluations performed are loaded into memory and combined with snapshots recorded by browser engines – Fig. 8 shows a fragment of some visual counterexamples generated.

7.3 Discussion & Limitations

We believe to have demonstrated that by using our framework, powerful reasoning can be supported over arbitrary web pages. From our experience, capturing such realistic regulatory requirements is not trivial, especially taking into account the perspective of practitioners that would use our approach. Specification aids such as patterns [36] or requirements elicitation and reasoning techniques [20] would help in supporting the methodology – something which is a typical concern with formal methods in practice. Extending such approaches to the web UI context appears quite promising. The page analyzed for WCAG requirements – the New York Times homepage – satisfied most of them. Somewhat surprisingly though, in certain cases the target page was not meeting the stated WCAG requirements, as illustrated in Fig. 8.

While an exhaustive WCAG 2.1 characterization was out of the scope of this paper, a meaningful portion of it has been formalized and analyzed over real-world pages and interactions. In terms of the capabilities of the STREL logic for expressing the *runtime* requirements of interest, we found it to be reasonably powerful. Most of the other rules of the specification fall into the following categories: (i) minor variations of the analyzed ones, (ii) *programmatically-determined* (i.e., analyzable by scanning the source code of the page, without the need of a runtime evaluation of the code) and (iii) media-related (i.e., relating image, video, audio content, and therefore out of our scope). However, there are cases where an exactly equivalent specification required some extra machinery (e.g., in Formula 14, that required the special $\sim$ comparator), or was unnecessarily complicated (e.g., Formula 15 has a rough approximation of "flashing" as changing color within the subsequent half-second). It is worth noting that while the proposed approach and tool are presented on a two-dimensional model and make the hypothesis of a classical flat screen, there is in principle no limitation in considering more dimensions and unusual layouts (e.g., like the ones observable in virtual or augmented reality).

Regarding performances (rightmost columns of Table 1), we observe that both the computational time and memory usage seem suitable for the integration of complex specifications in a development flow. A notable difference we observe in this regard comes from that adoption of spatial operators (e.g., in the reflow property), which can have a significant impact on the overall computational time, in line with the literature on spatio-temporal logics [38], but still resulting in reasonable times. We note that the spatio-temporal models generated are quite sizable since each pixel corresponds to a spatial node, yielding evaluation models with sizes in the order of hundreds of thousands of nodes (e.g., 480K nodes and 2M edges for the 800x600 model). Recall that models are verified in an explicit-state way. Nevertheless, with an increase in the number of nodes of 265% (from the mobile resolution to the desktop analyzed), negligible differences can be observed between mobile and desktop memory usage and computational time. This high efficiency in the analysis is the primary benefit of exploiting the available algorithms for monitoring STREL formulae. Similarly, the high regularity of the spatial model (a regular, homogeneous grid) allows for a high level of data locality during computation, which is unprecedented in the traditional literature on STREL monitoring. A sizable memory penalty comes from tracking multiple events for a given session. This is not surprising, as any event triggers a snapshot of the state of the page which is then expanded into a full explicit-state representation of the observed data, and possible counter-examples. Conversely, execution time is not significantly affected. Therefore, when explicit counter-example generation is not needed, complex event traces can be handled efficiently.

Concerning the number of affected elements and events (center part of Table 1), we observe differences when a given property is monitored on mobile versus desktop. This is because the analyzed website involves a significant number of decisions at runtime about the content to show depending on the resolution of the navigating browser. We believe this motivates – even more – the adoption of a logic-based specification language as it traditionally gives room to a large variety of

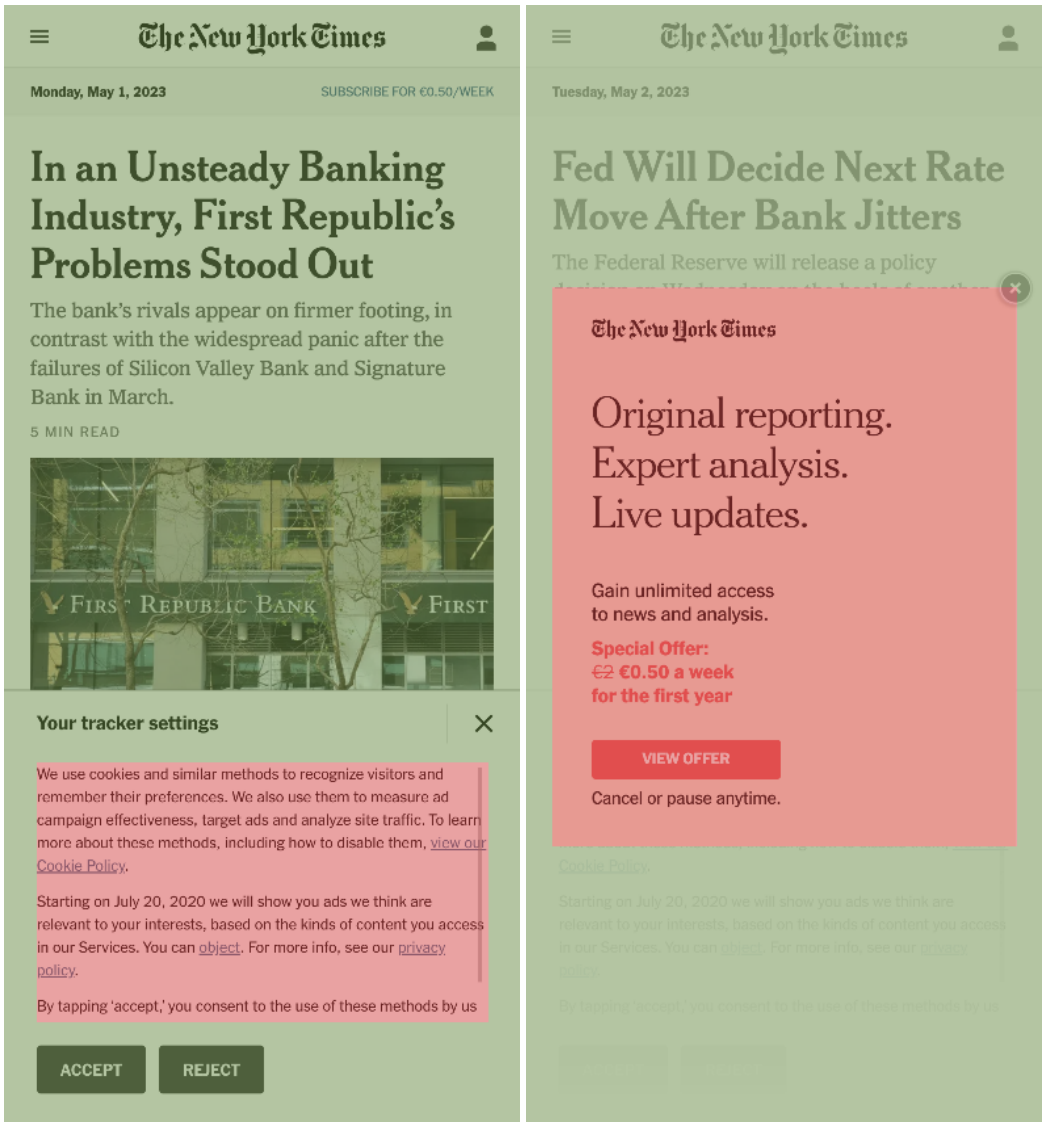

Fig. 8. Visual counterexamples automatically generated following evaluation of Formula 12 (on the left) and Formula 16 (on the right) for the New York times homepage for a mobile screen. The counterexample shows in red areas where the formula yields `false` (the property is violated), and in green areas where the formula evaluation yields `true`. In this case, the violation comes from the fact that the text block is too long, resulting in unwanted cuts of the text (left), or in the popup disappearing despite the user focusing it (right).

pre-condition definitions. Furthermore, a remark must be noted about the intrinsic effects of general selectors (e.g., `button`) vs specific ones (e.g., `.secondaryNav : nth – child(1)`), and about simple formulae (with 1-3 operators) vs complex ones (4+, mixing spatial and temporal). We noted that general selectors and complex formulae often result in some performance penalty, but importantly, they typically significantly reduce the understandability of counterexamples: it becomes harder to

determine which element of the page (or part of the formula) is failing. On the other hand, they typically correspond to more reusable and more compliant specifications, so this trade-off might lead to different preferences depending on the user (e.g., external evaluator vs developers).

In conclusion, the advantages of our formal perspective in these kinds of specifications are very appealing, as they can be defined and monitored regardless of the specific physical device being used, the web framework adopted, or the actual browser that is running the browsing session – any browser that implements the W3C WebDriver API is compatible. That said, a consideration about the usability of our tool must be made: we provide a DSL that makes the adoption quite straightforward, thanks to the CSS notation that we use for the atoms (which is familiar for web developers), to the minimal syntax we introduce, and to the advanced tooling that complements WEBMONITOR. Nevertheless, at this stage a background in formal methods can certainly speed up the adoption for a developer, although we believe that the convenience of automating a large part of the translation from the specification to the testing can be very appealing and motivate the effort. The idea of a logic language that is close to natural language is also to support developments in the direction of specification synthesis, possibly driven by state-of-the-art natural language processing techniques, as briefly discussed in Sec. 6.3, where plain-English requirements are translated to a subset of formulae of the logic language we are targeting.

8 RELATED WORK

This work introduces a novel approach for automating the analysis of graphical user interfaces by monitoring spatio-temporal specifications. While the work can naturally be positioned along the line sketched by the theoretical results of [47] that systematically analyzes formally web pages, our focus is not on the document structure, but on its visual representation, or user interface (UI). Recent approaches to UI testing, whether performed by exact match, like in [62], in Sikuli [61], or by learning methodologies [18, 27] and genetic algorithms [33], are primarily analyzing image renders of the pages. While image analysis can provide great insights regarding some accessibility aspects like contrast ratios, it is not very helpful in cases where complex interaction patterns are present or when analyzing a broad set of resolutions. Recent approaches to formal analysis of UIs do not encompass runtime information. In [45], the authors use a similar browser-engine-based approach to derive value bounds for a given attribute; in [63] from static analysis abstract traces are derived, while [44] and [46] follow a component-based approach, avoiding to check the page as a whole. In contrast to them, our work advocates browser-in-the-loop and is agnostic about the component approach adopted by the developers. From the perspective of automated software testing, instead, well-established tools like Selenium [10] and Cypress [43] provide a wide range of features to support the precise crafting of UI unit tests and multiple browsers[19]. However, they are not as effective when integrating different components. Compared to these, our approach is on a different abstraction level: it allows to express more abstract requirements (e.g., requirements for any paragraph) at the expense of a higher computational cost.

Spatial (and spatio-temporal) logics historically emerged in the context of models of the physical space [54], and STREL [38] is no exception. Alternative logics could be considered. SLCS [12] proposes a topological approach, which allows for more abstract specifications. Conversely, TSSL [5] introduces a more efficient quad tree-based modeling that could suit well the regular grids we are addressing, despite, in that case, it serves the description of temporal evolutions. More expressive logics like spatial μ-calculus [32] could be a valuable alternative, although the computational penalty might be prohibitive. Nevertheless, we claim that STREL provides a good balance between expressivity (for the kinds of specifications we analyzed) and performance.

The declarative approach of logic-based specification of WEBMONITOR can resemble the one developers are accustomed to in behaviour-driven-development (BDD [42]) and related approaches

to acceptance testing. The idea of BDD is to write executable specifications in natural language, organized in some fixed structure. The issue with BDD though, is that the translation to executable code is arbitrary and not necessarily consistent among the specifications. Approaches like [30] try to automate part of the process by translating parts of the specification to temporal automata. Similarly, [28] supports the formalization effort by enriching the BDD with semi-formal graphs developed by users supported by experts. [52], instead, explores the idea of automatically generating variations of a specification given a manually written one, as a form of lightweight model-checking. Our approach looks to be quite in the middle of the spectrum between BDD and classical logic specification: the user writes specifications in a richer DSL, but does not need to write any line of glue code to transform the specification in executable code. Monitoring is a prominent runtime verification technique, where a verdict is expressed by evaluating data traces describing single execution against a formal specification. Moonlight [6] is a popular tool for spatio-temporal monitoring, which also supports parallel execution of the monitors, a valuable option when in a multi-threaded execution environment. Several alternatives could be considered. Since STREL is an extension of Signal Temporal Logic, typical STL [34] monitors could be exploited, in particular *lead/lag-bounded transducers* [35], RTAMT [40], and Breach [16]. However, we point out that while those tools can provide efficient evaluation for temporal traces, they are not developed taking into account evaluations depending on a spatial model, and would therefore require a significant amount of work to reach comparable performances and expressivity. Alternatives worth mentioning are the ones in the context of field-calculus [4, 48]. However, no comparison between spatio-temporal monitor performances is available because of the different languages they address.

Systematically testing the accessibility of graphical user interfaces is a topic that has become more prominent in the latest years, with several works [7, 8, 23] that have addressed the problem from practical use-cases perspective. Several tools have emerged for this purpose. [51] made a preliminary survey, although not focused on the web specifically. The W3C maintains a curated list of accessibility evaluation tools [58]. Very recently, [25] established a set of requirements they should satisfy, many of which are in line with the goal of WebMonitor. Finally, the adoption of formal methods in CI/CD pipelines has not been extensively applied, although some use cases where its adoption has proved to be beneficial are presented in [11, 29]. Conversely, automated UI testing in CI/CD pipelines is gaining interest [3], as tools for automating the interaction like TestComplete [2] grow in maturity.

9 CONCLUSION AND FUTURE WORK

Guided by the crucial role that graphical user interfaces play for contemporary web platforms, we introduced a novel approach to formalize and evaluate their spatio-temporal behaviors. We presented the formal foundations and the development of the WebMonitor tool, a software framework for automatic monitoring of STREL specifications of web pages, which encapsulates Moonlight and Selenium WebDriver for the formal aspects and the interaction with the Web, respectively. A continuous integration script and a facility producing visual counterexamples of requirement violations assist the developer within the development workflow. We evaluated our approach over a real-world target whereupon automated monitoring of requirements sourced from the latest WCAG2.1 standards was performed, illustrating its applicability. Finally, we investigated the verification performance and possible advantages and disadvantages of a formal approach to web specifications evaluation. The formal approach advocated in this paper is independent of the underlying technologies a web application is developed with (e.g. React, PHP, or other) , as well as from the browser/operating system in use – a stark difference from the state-of-the-art testing libraries.

We identify several avenues for future investigation. A first interesting direction is the one of symbolic methods developed for traditional model checking [9], which could be adopted for the

spatial fragment. Along the same line, a comparison of the expressivity with symbolic-based approaches could give better insights about the benefits and limitations of the approach. It would also be interesting, in future works, to connect with the literature on page interaction, to provide an automated end-to-end solution for pages testing. In terms of the performances, several possible optimizations could be explored: firstly, the high level of spatial locality observed from the specifications over UIs of this work might suggest (i) the adoption of more efficient algorithms, parallelizing the execution over "far" areas of the screen, and (ii) efficient data structures can be used for compressing large areas where a property is satisfied or violated (e.g. coarser grids of multiple pixels could be used to cut computational time, or deciding the granularity of the grid at runtime, by analyzing the specification to monitor). Additionally, another performance boost could come from more clever tessellations of the spatial model, by changing it dynamically depending on the specification one wants to monitor. Another possibility could be to have algorithms optimized to perform a lazy evaluation of formulae maximizing the evaluation throughput so that a branch of the formula is only evaluated when the other is not sufficient for providing a final verdict, or property-driven model slicing [55] can be pursued. For the accessibility use-case investigated, perhaps alternative spatial logics could prove to be more efficient without significant costs in terms of expressivity. Also, in the spirit of Google's User Experience (UX) *web vitals* [22], formal spatio-temporal reasoning can be adopted to precisely quantify ranking metrics or alternative, developer-defined criteria. Lastly, while WEBMONITOR can already be used in practice, a future investigation should assess how developers interact with it, and collect crucial feedback to guide further improvement.

ACKNOWLEDGMENTS

This research has been partially supported by the Hellenic Foundation for Research and Innovation – Project 15706 RV4THINGS, by the Austrian Science Fund (FWF) for the project "High-dimensional statistical learning: New methods to advance economic and sustainability policies" (ZK 35), by MUR PRIN project 20228FT78M DREAM (modular software design to reduce uncertainty in ethics-based cyber-physical systems) and the consortium iNEST (Interconnected North-Est Innovation Ecosystem) funded by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS_00000043).

REFERENCES

[1] Ayush Agrawal, Siddhartha Gadgil, Navin Goyal, Ashvni Narayanan, and Anand Tadipatri. 2022. Towards a Mathematics Formalisation Assistant using Large Language Models. *ArXiv* abs/2211.07524 (2022), 57 pages. https://api.semanticscholar.org/CorpusID:253510131

[2] Samer Al-Zain, Derar Eleyan, and Joy Garfield. 2012. Automated User Interface Testing for Web Applications and TestComplete. In *Proceedings of the CUBE International Information Technology Conference* (Pune, India) *(CUBE '12)*. Association for Computing Machinery, New York, NY, USA, 350–354. https://doi.org/10.1145/2381716.2381782

[3] E. Alegroth, A. Karlsson, and A. Radway. 2018. Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 172–181. https://doi.org/10.1109/ICST.2018.00026

[4] Giorgio Audrito and Gianluca Torta. 2021. Towards Aggregate Monitoring of Spatio-Temporal Properties. In *Proceedings of the 5th ACM International Workshop on Verification and MOnitoring at Runtime EXecution* (Virtual, Denmark) *(VORTEX 2021)*. Association for Computing Machinery, New York, NY, USA, 26–29. https://doi.org/10.1145/3464974.3468448

[5] Ezio Bartocci, Ebru Aydin Gol, Iman Haghighi, and Calin Belta. 2018. A Formal Methods Approach to Pattern Recognition and Synthesis in Reaction Diffusion Networks. *IEEE Transactions on Control of Network Systems* 5, 1 (2018), 308–320. https://doi.org/10.1109/TCNS.2016.2609138

[6] Ezio Bartocci, Luca Bortolussi, Michele Loreti, Laura Nenzi, and Simone Silvetti. 2020. MoonLight: A Lightweight Tool for Monitoring Spatio-Temporal Properties. In *Runtime Verification*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.). Springer International Publishing, Cham, 417–428.

[7] Sebastian Bauersfeld and Tanja E. J. Vos. 2014. User Interface Level Testing with TESTAR; What about More Sophisticated Action Specification and Selection?. In *Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014, L'Aquila, Italy, 9-11 July 2014 (CEUR Workshop Proceedings, Vol. 1354)*, Davide Di Ruscio and Vadim Zaytsev (Eds.). CEUR-WS.org, L'Aquila, Italy, 60–78. https://ceur-ws.org/Vol-1354/paper-06.pdf

[8] Donald Beaver. 2020. Applied Awareness: Test-Driven GUI Development using Computer Vision and Cryptography. *CoRR* abs/2006.03725 (2020), 9 pages. arXiv:2006.03725 https://arxiv.org/abs/2006.03725

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Rance Cleaveland (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207.

[10] Andreas Bruns, Andreas Kornstadt, and Dennis Wichmann. 2009. Web Application Tests with Selenium. *IEEE Software* 26, 5 (2009), 88–91. https://doi.org/10.1109/MS.2009.144

[11] Tien-fu Chang, Alejandro Danylyzsn, So Norimatsu, Jose Rivera, David Shepard, Anthony Lattanze, and James Tomayko. 1997. "Continuous Verification" in Mission Critical Software Development. In *Proceedings of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Volume 5 (HICSS '97)*. IEEE Computer Society, USA, 273.

[12] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. 2014. Specifying and Verifying Properties of Space. In *Theoretical Computer Science*, Josep Diaz, Ivan Lanese, and Davide Sangiorgi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–235.

[13] European Commission. 2021. Web Accessibility Directive — Standards and harmonisation. https://digital-strategy.ec.europa.eu/en/policies/web-accessibility-directive-standards-and-harmonisation. Accessed: 2022-03-30.

[14] California Senate Judiciary Committee et al. 2018. California Consumer Privacy Act: AB 375 Legislative History.

[15] Michael Cooper, Alastair Campbell, Joshue O'Connor, and Andrew Kirkpatrick. 2018. *Web Content Accessibility Guidelines (WCAG) 2.1*. W3C Recommendation. W3C. https://www.w3.org/TR/2018/REC-WCAG21-20180605/.

[16] Alexandre Donzé, Thomas Ferrère, and Oded Maler. 2013. Efficient Robust Monitoring for STL. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 264–279.

[17] Matthew B. Dwyer, Vicki Carr, and Laura Hines. 1997. Model Checking Graphical User Interfaces Using Abstractions. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Zurich, Switzerland) *(ESEC '97/FSE-5)*. Springer-Verlag, Berlin, Heidelberg, 244–261. https://doi.org/10.1145/267895.267914

[18] Juha Eskonen, Julen Kahles, and Joel Reijonen. 2020. Automating GUI Testing with Image-Based Deep Reinforcement Learning. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, Piscataway, New Jersey, Stati Uniti, 160–167. https://doi.org/10.1109/ACSOS49614.2020.00038

[19] Boni García, Mario Munoz-Organero, Carlos Alario-Hoyos, and Carlos Delgado Kloos. 2021. Automated driver management for Selenium WebDriver. *Empirical Software Engineering* 26, 5 (23 Jul 2021), 107. https://doi.org/10.1007/s10664-021-09975-3

[20] Dimitra Giannakopoulou, Anastasia Mavridou, Julian Rhein, Thomas Pressburger, Johann Schumann, and Nija Shi. 2020. Formal requirements elicitation with FRET. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020) (Aeronautics ARC-E-DAA-TN77785)*. National Aeronautics and Space Administration (NASA), Langley Research Center, Hampton VA 23681-2199, USA, 6 pages.

[21] Daniel Glazman, John Williams, Tantek Çelik, Peter Linss, Ian Hickson, and Elika Etemad. 2018. *Selectors Level 3*. W3C Recommendation. W3C. https://www.w3.org/TR/2018/REC-selectors-3-20181106/.

[22] Ilya Grigorik. 2020. Introducing Web Vitals: essential metrics for a healthy site. blog.chromium.org/2020/05/introducing-web-vitals-essential-metrics.html. Accessed: 2021-12-16.

[23] Michael D. Harrison, Paolo Masci, and José C. Campos. 2019. Verification Templates for the Analysis of User Interface Software Design. *IEEE Transactions on Software Engineering* 45, 8 (2019), 802–822. https://doi.org/10.1109/TSE.2018.2804939

[24] Jie He, Ezio Bartocci, Dejan Ničković, Haris Isakovic, and Radu Grosu. 2022. DeepSTL: From English Requirements to Signal Temporal Logic. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 610–622. https://doi.org/10.1145/3510003.3510171

[25] Nicola Iannuzzi, Marco Manca, Fabio Paternò, and Carmen Santoro. 2022. Usability and transparency in the design of a tool for automatic support for web accessibility validation. *Universal Access in the Information Society* 1 (24 Nov 2022), 20 pages. https://doi.org/10.1007/s10209-022-00948-x

[26] ISO Central Secretary. 2019. *Ergonomics of human-system interaction —- Part 210: Human-centred design for interactive systems*. Standard ISO 9241-210:2019. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/77520.html

[27] Kateryna Ivanova, Galyna V. Kondratenko, Ievgen V. Sidenko, and Yuriy P. Kondratenko. 2020. Artificial Intelligence in Automated System for Web-Interfaces Visual Testing. In *Proceedings of the 4th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2020). Volume I: Main Conference, Lviv, Ukraine, April 23-24, 2020 (CEUR Workshop Proceedings, Vol. 2604)*, Vasyl Lytvyn, Victoria Vysotska, Thierry Hamon, Natalia Grabar, Natalia Sharonova, Olga Cherednichenko, and Olga Kanishcheva (Eds.). CEUR-WS.org, Lviv, Ukraine, 1019–1031. https://ceur-ws.org/Vol-2604/paper68.pdf

[28] Benjamin Weyers Judy Bowen and Bowen Liu. 2023. Creating Formal Models from Informal Design Artefacts. *International Journal of Human-Computer Interaction* 39, 15 (2023), 3141–3158. https://doi.org/10.1080/10447318.2022.2095833 arXiv:https://doi.org/10.1080/10447318.2022.2095833

[29] Henrik Kaijser, Henrik Lönn, Peter Thorngren, Johan Ekberg, Maria Henningsson, and Mats Larsson. 2018. Towards Simulation-Based Verification for Continuous Integration and Delivery. In *ERTS 2018 (9th European Congress on Embedded Real Time Software and Systems (ERTS 2018))*. Hyper Articles en Ligne, Toulouse, France, 10 pages. https://hal.archives-ouvertes.fr/hal-02156371

[30] Eun-Young Kang and Thiago Rocha Silva. 2023. Towards Formal Verification of Behaviour-Driven Development Scenarios Using Timed Automata. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. 612–616. https://doi.org/10.1109/APSEC60848.2023.00081

[31] Michael Kretschmer, Jan Pennekamp, and Klaus Wehrle. 2021. Cookie Banners and Privacy Policies: Measuring the Impact of the GDPR on the Web. *ACM Trans. Web* 15, 4, Article 20 (jul 2021), 42 pages. https://doi.org/10.1145/3466722

[32] Alberto Lluch Lafuente, Michele Loreti, and Ugo Montanari. 2015. A Fixpoint-Based Calculus for Graph-Shaped Computational Fields. In *Coordination Models and Languages*, Tom Holvoet and Mirko Viroli (Eds.). Springer International Publishing, Cham, 101–116.

[33] Gentiana Ioana Latiu, Octavian Augustin Cret, and Lucia Vacariu. 2016. *Automated Graphical User Interface Testing Framework—Evoguitest—Based on Evolutionary Algorithms*. Springer International Publishing, Cham, 39–63. https://doi.org/10.1007/978-3-319-23392-5_3

[34] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Yassine Lakhnech and Sergio Yovine (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 152–166.

[35] Konstantinos Mamouras and Zhifu Wang. 2020. Online Signal Monitoring With Bounded Lag. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3868–3880. https://doi.org/10.1109/TCAD.2020.3013053

[36] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. 2021. Specification Patterns for Robotic Missions. *IEEE Trans. Software Eng.* 47, 10 (2021), 2208–2224. https://doi.org/10.1109/TSE.2019.2945329

[37] John Mueller. 2008. *Accessibility for everybody: Understanding the Section 508 accessibility requirements*. Apress, London, United Kingdom.

[38] Laura Nenzi, Ezio Bartocci, Luca Bortolussi, and Michele Loreti. 2022. A Logic for Monitoring Dynamic Networks of Spatially-distributed Cyber-Physical Systems. *Log. Methods Comput. Sci.* 18, 1 (2022), 30 pages. https://doi.org/10.46298/lmcs-18(1:4)2022

[39] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (mar 2015), 66–73. https://doi.org/10.1145/2699417

[40] Dejan Ničković and Tomoya Yamaguchi. 2020. RTAMT: Online Robustness Monitors from STL. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 13 pages.

[41] JAKOB NIELSEN. 1993. Chapter 5 - Usability Heuristics. In *Usability Engineering*, JAKOB NIELSEN (Ed.). Morgan Kaufmann, San Diego, 115–163. https://doi.org/10.1016/B978-0-08-052029-2.50008-5

[42] Dan North. 2006. *Introducing BDD*. Better Software Magazine. https://dannorth.net/introducing-bdd/ Accessed: 2019-03-18.

[43] Narayan Palani. 2021. *Automated Software Testing with Cypress*. Auerbach Publications, Milton Park, Oxfordshire, United Kingdom. https://doi.org/10.1201/9781003145110

[44] Pavel Panchekha, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2019. Modular Verification of Web Page Layout. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 151 (oct 2019), 26 pages. https://doi.org/10.1145/3360577

[45] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying That Web Pages Have Accessible Layout. *SIGPLAN Not.* 53, 4 (jun 2018), 1–14. https://doi.org/10.1145/3296979.3192407

[46] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. *SIGPLAN Not.* 51, 10 (oct 2016), 181–194. https://doi.org/10.1145/3022671.2984010

[47] Seongbin Park. 1998. Structural Properties of Hypertext. In *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space—Structure in Hypermedia Systems: Links, Objects, Time and Space—Structure in Hypermedia Systems* (Pittsburgh, Pennsylvania, USA) *(HYPERTEXT '98)*. Association for Computing Machinery, New York, NY, USA, 180–187. https://doi.org/10.1145/276627.276647

[48] Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: Practical Aggregate Programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. Association for Computing Machinery, New York, NY, USA, 1846–1853. https://doi.org/10.1145/2695664.2695913

[49] Jenny Ruiz, Estefanía Serral, and Monique Snoeck. 2021. Unifying functional User Interface design principles. *International Journal of Human–Computer Interaction* 37, 1 (2021), 47–67.

[50] Esteban Sánchez and José A. Macías. 2019. A set of prescribed activities for enhancing requirements engineering in the development of usable e-Government applications. *Requirements Engineering* 24, 2 (01 Jun 2019), 181–203. https://doi.org/10.1007/s00766-017-0282-x

[51] Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. A Survey on the Tool Support for the Automatic Evaluation of Mobile Accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-Exclusion* (Thessaloniki, Greece) *(DSAI 2018)*. Association for Computing Machinery, New York, NY, USA, 286–293. https://doi.org/10.1145/3218585.3218673

[52] Colin Snook, Thai Son Hoang, Dana Dghyam, Michael Butler, Tomas Fischer, Rupert Schlick, and Keming Wang. 2018. Behaviour-Driven Formal Model Development. In *Formal Methods and Software Engineering*, Jing Sun and Meng Sun (Eds.). Springer International Publishing, Cham, 21–36.

[53] Simon Stewart and David Burns. 2018. *WebDriver*. W3C Working Draft. W3C. https://www.w3.org/TR/webdriver/.

[54] Christos Tsigkanos, Timo Kehrer, and Carlo Ghezzi. 2017. Modeling and verification of evolving cyber-physical spaces. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 38–48. https://doi.org/10.1145/3106237.3106299

[55] Christos Tsigkanos, Nianyu Li, Zhi Jin, Zhenjiang Hu, and Carlo Ghezzi. 2021. Scalable multiple-view analysis of reactive systems via bidirectional model transformations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 993–1003. https://doi.org/10.1145/3324884.3416579

[56] European Union. 2016-05-04. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal L110* 59 (2016-05-04), 1–88.

[57] Ennio Visconti, Christos Tsigkanos, and Laura Nenzi. 2023. WebMonitor: Verification of Web User Interfaces. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 170, 4 pages. https://doi.org/10.1145/3551349.3559538

[58] W3C Web Accessibility Initiative (WAI). 2023. Web Accessibility Evaluation Tools List. https://www.w3.org/WAI/ER/tools/. [Online; accessed 5-May-2023].

[59] Web Hypertext Application Technology Working Group (WHATWG) 2022. *HTML*. Web Hypertext Application Technology Working Group (WHATWG). https://html.spec.whatwg.org/multipage/webappapis.html#event-loop Living Standard.

[60] Yuan Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. 2023. Harnessing the Power of Large Language Models for Natural Language to First-Order Logic Translation. *ArXiv* abs/2305.15541 (2023), 16 pages. https://api.semanticscholar.org/CorpusID:258888128

[61] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology* (Victoria, BC, Canada) *(UIST '09)*. Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/1622176.1622213

[62] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and Image Recognition Driving Cross-Platform Automated Mobile Testing. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, Piscataway, New Jersey, Stati Uniti, 1561–1571. https://doi.org/10.1109/ICSE43902.2021.00139

[63] Zhen Zhang, Yu Feng, Michael D. Ernst, Sebastian Porst, and Isil Dillig. 2021. Checking Conformance of Applications against GUI Policies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/3468264.3468561