WebMonitor: Verification of Web User Interfaces

Ennio Visconti
TU Wien
Vienna, Austria

Christos Tsigkanos
University of Bern
Bern, Switzerland

Laura Nenzi
University of Trieste
Trieste, Italy

ABSTRACT

Application development for the modern Web involves sophisticated engineering workflows which include user interface aspects. Those involve Web elements typically created with HTML/CSS markup and JavaScript-like languages, yielding Web documents. WebMonitor leverages requirements formally specified in a logic able to capture both the layout of visual components as well as how they change over time, as a user interacts with them. Then, requirements are verified upon arbitrary web pages, allowing for automated support for a wide set of use cases in interaction testing and simulation. We position WebMonitor within a developer workflow, where in case of a negative result, a visual counterexample is returned. The monitoring framework we present follows a black-box approach, and as such is independent of the underlying technologies a Web application may be developed with, as well as the browser and operating system used.
WebMonitor **is available as open source software:**
https://github.com/ennioVisco/webmonitor
Video demonstration of WebMonitor**:**
https://youtu.be/hqVw0JU3k9c

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; • **Theory of computation** → *Logic and verification.*

KEYWORDS

Web Technologies, Spatio-temporal Logic, Runtime Verification

ACM Reference Format:
Ennio Visconti, Christos Tsigkanos, and Laura Nenzi. 2022. WebMonitor: Verification of Web User Interfaces. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3559538

1 INTRODUCTION

Application development for the modern Web is a sophisticated process that involves long engineering pipelines spanning from the specification of business requirements to the worldwide delivery of applications. The development of modern continuous integration (CI) and continuous delivery (CD) workflows has allowed developers to drastically automate large portions of these processes, by building complex pipelines. While most of the tasks involved in the process have seen a significant boost in the level of automation and repeatability in recent years, things are fairly different when considering User Experience (UX) issues. In fact, aspects like usability, accessibility, understandability, etc. are becoming increasingly crucial for modern applications, and significant industrial effort is being put into prioritizing applications that assess them at some level [3].

Validation within a user interface engineering workflow is dominated by User Interface (UI) testing, which typically entails inspecting visual elements to check that they are functioning according to requirements – both in terms of functional (such as regulatory conformance), and in terms of non-functional (e.g., performance, timings). Current Web UI testing 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 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.

To this end, we present WebMonitor, for monitoring Web UIs. In our approach, requirements are formally specified in a Domain-Specific Language (DSL) over a logic language able to capture two essential aspects of web documents; the layout of visual components and how they change over time, as a user interacts with them. The cornerstone of WebMonitor is that UIs can be formalized as a spatio-temporal model over locations of the graphical device (e.g. pixels of the screen). Over such a model, graphical UI components (e.g. buttons, images, input fields, etc.) are overlayed, while its temporal dimension corresponds to interaction events that a user may induce. The advantages of such a formal perspective are that (i) it can be defined and monitored regardless of the specific physical device or browser being used, and (ii) automated support can be enabled for a wide set of use cases in interaction testing and simulation.

In the following, we illustrate how Web documents can give rise to formal models, whereupon a spatio-temporal logic can be used to express requirements, presenting WebMonitor– an end-to-end technical framework, where given a webpage target, automated procedures carry out analysis processes.

Motivating example. Consider a cookie consent notice – a banner informing the user about the cookies stored on the browser to track a website's usage. Such a cookie popup should be able to be dismissed by the user easily. Specifically:

ER1. The popup should be *visible*; specifically, the entirety of the popup should be within the window that the user perceives as the interface so that all the relevant information is available.

ER2. The popup should be visible *within 2 seconds* after the page has been loaded and should *remain visible* until the user explicitly dismisses it.

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3559538

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 with some user action but also denotes timing constraints. In the following, we demonstrate how such requirements can be specified and monitored against a sample web page, illustrating WebMonitor capabilities. We note that *any* page can be used; the example page is adopted to simplify presentation of the specification features.

2 MODELS AND REASONING IN WEB UIS

Graphical user interfaces assume the presence of a display for interacting with the user – this is the case for all platforms, including the Web-based. Therefore, an intuitive interpretation of the 2-D graphical space corresponds to a set of (x, y) coordinates that identify each element of the pixel-grid that composes the physical display of the targeted device. We devise a spatial model capturing all the logical locations pixels can occupy, identified by coordinate pairs.

Recall the running example; requirements entail checking the behavior, including any conditions that make a given component of the GUI get-in or get-out of the user's focus. More generally, in the context of the web, three conceptually different areas of the screen should be represented: (i) the **document**, which is the logical region over which the web page is defined throughout its lifecycle; (ii) the **layout viewport**, which represents the reachable area of the page, and (iii) the **visual viewport**, which is the region of the page that is displayed to the user at a given moment. A requirement like ER2 would translate to the popup being in the visual viewport until the user explicitly dismisses it. In practice, this is described by associating different values as propositions in each pixel, to express the presence in different areas of the screen (i-iii).

Following the same intuition, UI visual entities can be captured in the spatial model. For example, the outer box of a cookie popup could be defined by developers utilizing HTML as such:

$$< \texttt{div class} = \text{“}\texttt{cookieInfo}\text{”} > \texttt{We use cookies to...} < \texttt{/div} >. \quad (1)$$

However, first-class objects of a web UI are not only visual components such as the aforementioned cookie popup outer box; they include also styling properties as well as events, the latter corresponding to the user's interaction with the UI. Standard web APIs already provide a wide range of events fired by the browser engine when the user interacts with the page. Consider primary browser events such as *click*, *focus*, *scroll*, and *load*, fired respectively when the user clicks an element, selects an element, scrolls the page, and when the page loading is completed. These will be taken into account for reasoning over the *dynamics* of the interface as the user interacts with it. Such events fire in the browser's event loop, a standard HTML concept that guarantees synchronous recording of user events [4]. Back to our example, as the user interacts with the cookie popup, only a small set of events is recorded – initially there is the page *load*, which may then be followed by a series of *scroll* and *click* events. Behind the scenes, WebMonitor relies on a formal spatio-temporal model that captures visual and structural parts of the UI, as well as its evolution due to possible user actions. Using such a formal model, automated reasoning can be enabled, and properties reflecting the desired requirements can be evaluated over its spatio-temporal evolution.

To specify properties capturing UI requirements, we adopt a specification language predicating in both space (i.e., the visual UI) and time (i.e., the user's behavior). We work within a fragment of the *Spatio-Temporal Reach and Escape Logic* (STREL) [1], which we describe briefly over the running example. For a complete formal treatment and semantics of the logic, the interested reader is referred to [1]. Although we present logical formulae for conciseness, a DSL with this semantics is used for actual specification in WebMonitor. A STREL formula conforms to the following grammar:

$$\varphi := p \circ c \mid b \mid \quad (2)$$

$$\top \mid \bot \mid \neg \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \quad (3)$$

$$\mathrm{F}_{\leq t}\varphi \mid \mathrm{G}_{\leq t}\varphi \quad (4)$$

Recall that requirements ER1-2 are centered on 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. one 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.

$$\mu_{visible} := \texttt{.cookieInfo\$visibility} = \text{‘}\texttt{visible}\text{'} \quad (5)$$

Formula 5 encodes that the visibility property of all the elements of class `cookieInfo` equals to `visible`. To retrieve a specific element of the page in an atomic property, WebMonitor adopts the standard selector notation [2], followed by the special character '$' to denote the exact styling property that is being analyzed. An example of a 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, such property permits to specify the pixels that belong to the visual viewport. Back to our cookie popup example, the requirement (ER1) can be formalized as follows:

$$\phi_{ER1} := \mu_{visible} \wedge \texttt{screen}. \quad (6)$$

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

To predicate about behaviour, temporal operators (Formula 4) are used to express the fact that a subformula φ is satisfied *for some* (respectively *for all*) next time points t, as in usual temporal logics. To illustrate that, consider expressing the requirement (ER2):

$$\phi_{ER2} := \mathrm{F}_{\leq 2}(\phi_{ER1} \wedge (\texttt{button.close@active} \rightarrow \mu_{hidden}), \quad (7)$$

where

$$\mu_{hidden} := \texttt{.cookieInfo\$visibility} = \text{‘}\texttt{hidden}\text{'} \quad (8)$$

Formula 7 expresses the fact that ϕ_{ER1} should hold within 2 seconds: i.e., the popup should be visible to the user within 2 seconds, and, whenever the `button` element with class `close` is clicked (i.e. becomes active), then (→) it must become invisible ('`hidden`').

3 A WALK THROUGH WEBMONITOR

Having briefly illustrated how UI requirements can be specified into formal spatio-temporal properties over UI elements, WebMonitor can be used to evaluate them on an arbitrary webpage. WebMonitor contains automatic facilities for automating interfacing with browsers, generating spatio-temporal models upon which verification is performed, and subsequently interpreting analysis output

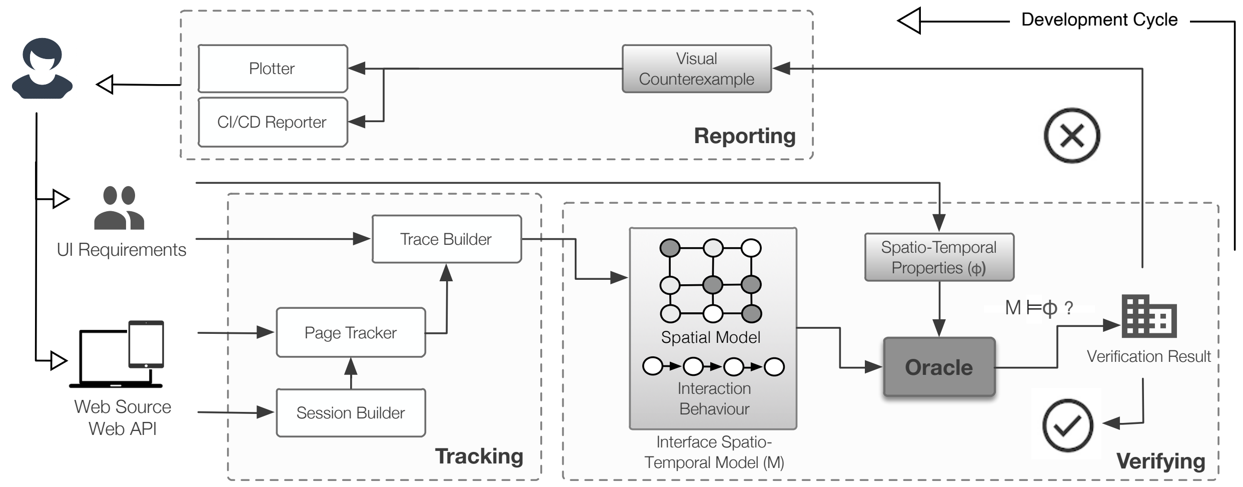


Figure 1: WebMonitor: Key components (white) and artifacts (shaded), over activities (dashed lines).

in a visual, meaningful way. Figure 1 shows a typical WebMonitor workflow highlighting the primary components, artifacts and activities involved. The overall process starts with a *Web Source*, which denotes a URL as well as auxiliary parameters required such as 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 primary input consists of the `Web Source` descriptor, which delimits the scope of the evaluation (e.g., browser sizes of interest, destination URL, browser engines of interest). The target descriptor is used to launch a browser session (via `Session Builder`) that is passed to the `Page Tracker` component, which is responsible for the interactions with standard Web APIs that will be used to fetch elements and events as described by the atomic propositions found in the specification. The concrete output of this stage is a spatio-temporal model of the Web UI at hand. Selenium Webdriver[1] APIs are leveraged for the interaction with web browsers, by using instructions that work interchangeably across browsers.
- *Verifying*: The second stage of the WebMonitor workflow is responsible for the actual analysis of the specification with respect to the data collected at the previous stage. An oracle component encapsulates a monitor to verify spatio-temporal properties, which evaluates the satisfaction of the specification. The monitoring facilities are provided by the Moonlight[2] 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 a violation *counterexample*, as in classical model checking. The developer may revise the design accordingly and restart the verification process. The second form of reporting functionality targets CI/CD pipelines. In this case, the `CI/CD 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), which can be used e.g., to decide whether or not to deploy the application in production.

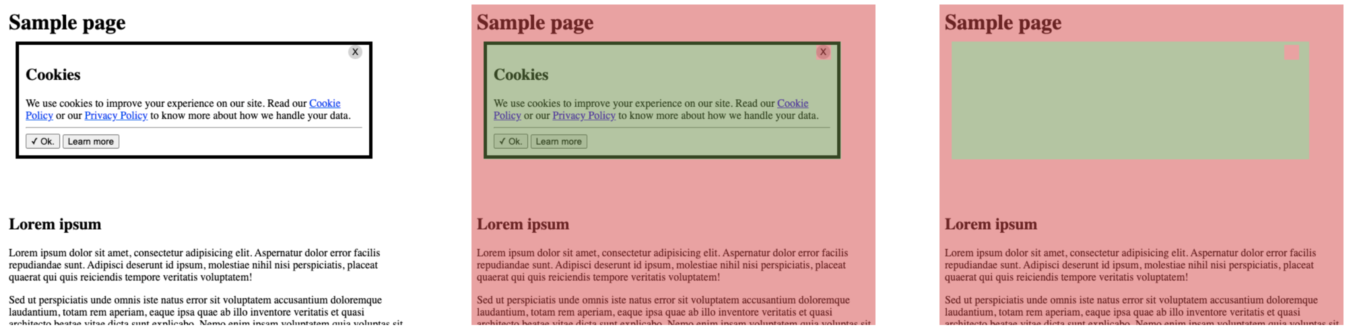
[1]Info at www.selenium.dev
[2]Info at github.com/MoonLightSuite/MoonLight

4 WEBMONITOR IN PRACTICE

Guided by the crucial role that graphical user interfaces play for contemporary web platforms, we introduced a novel tool to formalize and evaluate their spatio-temporal behaviors. WebMonitor is a software framework for automatic monitoring of spatio-temporal specifications of web pages, which encapsulates Moonlight and Selenium WebDriver for the formal verification aspects and for the interaction with the Web, respectively. A facility producing visual counterexamples of requirement violations assists the developer within the development workflow. We especially note that the formal approach advocated in this paper is independent from the underlying technologies a Web application is developed with, as well as from the browser and operating system in use.

WebMonitor is available as open source software, as a JVM-based software tool for reasoning on spatio-temporal behaviors of web pages. To use WebMonitor in practice, a developer follows four distinct steps:

(1) Initialization; a target web page is specified, along with parameters concerning browser and screen size, illustrated in Listing 1.
(2) Specification; requirements that the design should fulfill are specified in terms of the language outlined in Sec. 2, which is encoded by the WebMonitor DSL fragment shown in Listing 2.
(3) Analysis; verification facilities are invoked.
(4) Reporting; in case of violation, the visual counterexample is inspected.



Sample page

Cookies

We use cookies to improve your experience on our site. Read our Cookie Policy or our Privacy Policy to know more about how we handle your data.

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Aspernatur dolor error facilis repudiandae sunt. Adipisci deserunt id ipsum, molestiae nihil nisi perspiciatis, placeat quaerat qui quis reiciendis tempore veritatis voluptatem!

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit

(a) Initial state **(b) Evaluation on initial state** **(c) Evaluation after click**

Figure 2: Visual counterexamples returned as the evaluation result on a sample page as a web target for requirements ER1-2.

A typical user workflow is illustrated in Fig. 2, where the same page is evaluated over a trace composed of the loading of the page (Figure 2b), and the click of the closing button (Figure 2c). Red areas denote pixels of the pixel grid where ER2 *is* not satisfied, while the green one denote the areas where the property *is* satisfied.

Listing 1 provides the settings that have been used to define the web source for the analysis. `screenWidth` and `screenHeight` set the size of the browser window for the evaluation. Note that the optional parameter `browser`) allows selecting an alternative web browser engine, while the optional waiting time (`wait`) can be set to give some extra time before starting the evaluation, to be sure that the page is fully loaded. Lastly, `maxSessionDuration` and `targetUrl` set the maximum duration of the session and the URL of the web page under analysis, respectively.

Listing 1: WebSource definition.

```
WebSource.screenWidth = 600 // px
WebSource.screenHeight = 500 // px
WebSource.browser = Browser.CHROME
WebSource.wait = 0
WebSource.maxSessionDuration = 5_000 // ms
WebSource.targetUrl =
  "https://enniovisco.github.io/webmonitor/"
```

An example of the specification of requirements ER1-2 is shown in Listing 2, exploiting WebMonitor's Kotlin-based DSL. Three primary hooks are available for the developer: `Spec.atoms()` for defining the atoms of the logic, `Spec.record()` to express the events that fire a snapshot of the browsing session, and `Spec. formula` that defines the final specification to analyze. `select{}` is used to select HTML elements based on the passed query string, while `read` is an optional function that selects a specific parameter of the element, that is then compared for equality (by `equals`) or inequality (by `lessThan, lessThanEquals, ...`) to some value. Lastly, the `at` operator is optional to state that the given atom will be considered true only when that event is fired. `after{}` is used to state which events will be considered for firing new snapshots, while STREL operators are used directly inline as shown in the helper formulae fragment.

Listing 2: Specification definition.

```
Spec.atoms (
    select { ".cookieInfo" }  // [0]
            read "visibility"
            equals "visible",
    select { ".cookieInfo" }  // [1]
            read "visibility"
            equals "hidden",
    select { "button#close" } at "click" // [2]
)
Spec.record (
    after { "click" },
    after { "touch"}
)
// helper formulae
val screen = Spec.screen
val isVisible = Spec.atoms[0]
val isHidden = Spec.atoms[1]
val buttonClick = Spec.atoms[2]
val ER1 = isVisible and screen
val innerER2 = er1 and (buttonClick implies isHidden)
val ER2 = eventually(innerER2)
Spec.formula = ER2 // Final formula
```

Future directions on WebMonitor development will primarily be centered on expanding the DSL to support comparisons of complex data types (e.g. with operators like `isDarkerThan`), or a wider range of web features (e.g. the CSS pseudo-element `::before`); in addition, several performance enhancement options can be pursued, as well as more comprehensive counterexample reporting. An empirical assessment of the end-to-end approach implied by the tool and its integration within the contemporary web development process should be evaluated. DSL-specific aspects can further identify desirable specification features and abstractions.

REFERENCES

[1] Ezio Bartocci, Luca Bortolussi, Michele Loreti, and Laura Nenzi. 2017. Monitoring Mobile and Spatially Distributed Cyber-Physical Systems. In *Proc. of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design* (Vienna, Austria) *(MEMOCODE '17)*. Association for Computing Machinery, New York, NY, USA, 146–155. https://doi.org/10.1145/3127041.3127050

[2] Elika Etemad and Tab Atkins Jr. 2018. *Selectors Level 4*. W3C Working Draft. W3C. https://www.w3.org/TR/2018/WD-selectors-4-20181121/.

[3] 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.

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