Synthesizing Web Element Locators

Kartik Bajaj
University of British Columbia
Vancouver, BC, Canada
kbajaj@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
karthikp@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—To programmatically interact with the user interface of a web application, element locators are used to select and retrieve elements from the Document Object Model (DOM). Element locators are used in JavaScript code, Cascading stylesheets, and test cases to interact with the runtime DOM of the webpage. Constructing these element locators is, however, challenging due to the dynamic nature of the DOM. We find that locators written by web developers can be quite complex, and involve selecting multiple DOM elements. We present an automated technique for synthesizing DOM element locators using examples provided interactively by the developer. The main insight in our approach is that the problem of synthesizing complex multi-element locators can be expressed as a constraint solving problem over the domain of valid DOM states in a web application. We implemented our synthesis technique in a tool called LED, which provides an interactive drag and drop support inside the browser for selecting positive and negative examples. We find that LED supports at least 86% of the locators used in the JavaScript code of deployed web applications, and that the locators synthesized by LED have a recall of 98% and a precision of 63%. LED is fast, taking only 0.23 seconds on average to synthesize a locator.**

Keywords: Program synthesis, Programming by example, Element locators, CSS selectors, Web applications

I. Introduction

Over the past few years, the number of web applications has exploded. These web applications consist of three parts, namely 1) HTML code that is used to define the Document Object Model[1] (DOM), 2) CSS stylesheets that are used to define the layout of the web page, and 3) JavaScript code that interacts with and updates the DOM tree for the web application. DOM element locators define the rules required to traverse the DOM tree defined by the HTML. Both CSS and JavaScript utilize these rules and the information present in the DOM tree to interact with the elements present in it.

Writing DOM element locators is a complex task for developers as (1) they need a mental model of the valid DOM elements in each DOM state that they want to access, and (2) they need to find the common properties that are shared by the elements they want to access (to be inclusive), but also those that are not shared by any other element in the DOM state (to be exclusive). Developers typically perform this task by manually inspecting different elements of the DOM in the state of interest, and formulating DOM element locators through trial and error. This often leads to bugs and inconsistencies. For example, in our prior work [26], we found that 65% of the client-side JavaScript errors within a web applications are caused by faulty interactions between JavaScript and DOM.

In this paper, we present an automated approach to synthesize DOM element locators that can be used to select multiple DOM elements within JavaScript or /CSS code. We leverage *program synthesis* techniques to convert the problem of DOM element locator synthesis into a constraint solving problem. Our technique falls in the category of *Programming by example (PBE)*, which involve generating code based on input-output examples provided by the developer[23], [9], [14], In our case, the input is provided as positive and negative examples of DOM elements. We devise a mathematical model for translating the examples to constraints that are then fed into a Satisfiability (SAT) solver. The output of the solver is converted back to the DOM element locators that can be used as parameters to DOM API methods within JavaScript code. The synthesized code snippets are then ranked based on programmer-provided criteria and presented to the programmer. These can be used in JavaScript/CSS code that may be used within the web application to select the relevant DOM elements.

Our code synthesis technique is implemented in a tool called LED (Live Editor for DOM). LED has a visual interface that is interposed on the web application, allowing developers to interactively provide (i.e., drag and drop) examples of DOM elements they wish to select in a given DOM state. Developers can also augment the input with negative examples to further refine the synthesized code. LED also provides the developers the ability to configure the type of DOM element locators (such as minimum length, maximum length, selection scope) that they want to generate.

Prior work has explored the problem of constructing DOM element locators in JavaScript applications [1], [2], [5]. However, their focus has been on generating selectors for test cases in automated testing frameworks such as Selenium. The main difference between our work and these is that the work on test case generation is concerned with accessing a single element, while our work is concerned with selecting multiple DOM elements. This is an important difference as programmers often access multiple DOM elements with a single locator, as we show later in the paper. Another major difference between LED and prior work is that we not only capture the DOM elements required by the developer, but also the DOM elements that the developer does not to select. *To the*

[1]The Document Object Model is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents.

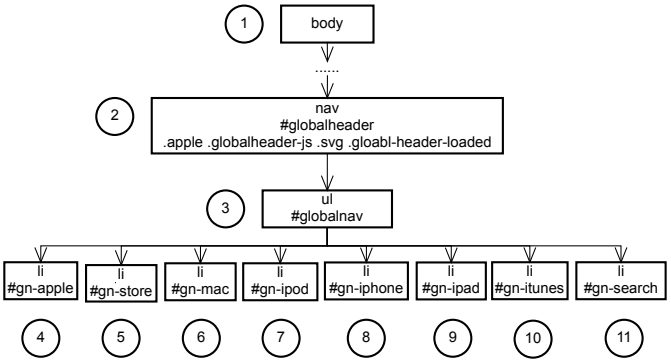

Fig. 2: Overview of the DOM structure for the running example.

```
1 var elems = document.querySelectorAll('#gn-store,#gn-mac,#gn-ipod,#gn-iphone,#gn-ipad,#gn-itunes');
2
3 for(var i=0; i<elems.length; i++) {
4   elems[i].addEventListener('onmouseover', 'changeColor');
5   elems[i].addEventListener('onmouseout','restoreColor');
6 }
7
8 function changeColor() {
9   //change element color here
10 }
11
12 function restoreColor() {
13   //restore element color here
14 }
```

Fig. 3: JavaScript code required to perform the sample task

TABLE I: Subset of types of DOM element locators available to developers

Type	Example	Description
`#id`	`#globalnav`	Select an element with id="globalnav".
`.class`	`.apple`	Select all elements with class="apple".
`*`	`*`	Select all DOM elements.
`element`	`li`	Select all elements.
`element,element`	`ul,li`	Select all and elements.
`element>element`	`ul>li`	Select all elements where the parent is a element.

best of our knowledge, we are the first to propose an automated approach for DOM element locator synthesis that can select multiple DOM elements based on examples provided by the programmer.

Our work makes the following contributions:

- A discussion of the problem of synthesizing JavaScript code that interacts with multiple DOM elements in one or more DOM states of the web application.
- An example based automated approach to analyze multiple DOM elements and generate constraints based on the provided examples, which are then input to a SAT solver.
- An implementation of our approach in an open source tool called LED.
- An empirical evaluation to validate the proposed technique, demonstrating its efficacy and real-word relevance. Our examination of real world web applications indicate that LED can support 86% of the total DOM element locators used by web developers. Further, developers often write complex locators to choose multiple DOM elements. The results of our empirical evaluation on three web applications show that LED can achieve a recall of 98% and precision of 92%, with a maximum execution time of 0.5 seconds.

II. Motivation

In this section, we first provide a running example to demonstrate the problems faced by web developers. We then discuss the challenges involved in synthesizing DOM element locators using our approach.

A. Running Example

Figure 1 presents the screenshot of a navigation menu from the Apple [3] website. The DOM tree structure of the menu is presented in Figure 2. Certain elements and class names have been omitted for the sake of simplicity. Each node within the DOM tree represents a DOM element. We use the CSS selector notation to present the DOM tree[2]. Words proceeded with '#' represent ID attributes, '.' represents classes attributes, and words without any prefixes represent the type of DOM element. Every DOM element will have a type associated with it, while class names and IDs are optional attributes. IDs are required to be unique within the DOM tree and can be used to refer to a specific DOM element.

Sample task. Assume the user needs to highlight the menu item elements, excluding the logo and search box, when the mouse hovers over the menu.

Solution. Figure 3 shows the JavaScript code required to perform the given task. At Line 1, we can see that the developer needs to pass a DOM element locator as a parameter to the DOM API function. The traditional way of writing DOM element locators involves the following steps; 1) load the web application, 2) navigate to a particular DOM state, 3) analyze the available DOM elements, and 4) write a DOM element locator using the information attached to each DOM element.

B. DOM element locators and Challenges

Table I provides a subset of types of DOM element locators that can be used by web developers to select the DOM elements. A complete list of types of DOM element locators can be found on the w3schools website [8]. Developers can use any of these DOM element locators depending upon the type of task as well as the information available in the DOM tree. Developers can also use additional information such as class names attached to each element in the DOM element locator.

For the running example, Table II lists a subset of the possible DOM element locators that can be used to select elements 5-10 in the Figure 2. However, all of the DOM

[2]We use the terms DOM element locators and CSS selectors interchangeably in the rest of this paper.

Fig. 1: Screenshot of navigation menu from Apple website

TABLE II: Subset of possible DOM element locators used to select elements 4-11

Locator	Locator Code	Locator Type	Count
1	`#gn-store, #gn-mac, #gn-ipod, #gn-iphone, #gn-ipad, #gn-itunes`	`#id`	8
2	`li`	`element`	1
3	`body nav ul li`	`element>element`	1
4	`body .apple ul li`	`element>element`	1
5	`body .globalheader-js li`	`element>element`	1
6	`#globalheader ul li`	`element>element`	1
7	`#globalheader li`	`element>element`	1
8	`#globalnav li`	`element>element`	1

element locators are not equally precise and durable. For example, locator 1 points to elements 5-10 only, whereas locators 2-8 point to additional elements including element 4 and 11 as well. Similarly, a DOM element locator that includes all selectors along the path from root node to target DOM element (e.g., locators 3,4), is precise, but fragile to minor modifications in the DOM tree. On the other hand, a DOM element locator that excludes the hierarchical information and only uses locators attached to target DOM element (e.g., locators 1,2) is resistant to DOM tree manipulations, but sometimes cannot precisely locate the target DOM element.

Therefore, writing a DOM element locator is a non-trivial task and requires a significant amount of effort by the developer. Further, since a single web application often involves multiple developers, each responsible for different parts of the application, the developer needs to carefully analyze and select the elements that she wants to interact with. This process is time consuming, and is a major source of errors for many web applications [26]. Despite all these issues, however, DOM element locators are heavily used by developers to obtain references to DOM elements within the JavaScript code. Therefore, the developers need automated tools that can 1) assist them in generating locators to obtain references to DOM elements, and 2) provide an immediate feedback on the quality of their locators used and the elements selected by that DOM element locator.

C. Goal

In this work, we present a technique to synthesize DOM element locators that may be used as parameters to DOM API methods within JavaScript code that interacts with DOM. The overall goal of this paper is to *assist the web developers in writing DOM element locators for JavaScript/CSS code that interacts with DOM*. In general, a good DOM element locator should satisfy the following criteria:

- The dependency of DOM element locator on the DOM hierarchy should be minimum, so that minor changes in the DOM should not invalidate the locator.
- The DOM element locator should be specific enough to narrow down the required DOM element(s) and avoid non-required DOM elements.

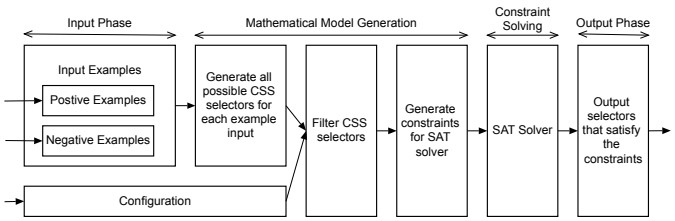

Fig. 4: DOM element locator code synthesis overview

To minimize the dependency of a DOM element locator on the DOM hierarchy while still preserving the preciseness of the selector, we need to rank the list of all selectors found in the DOM hierarchy and then combine only the most optimal selectors to generate the required DOM element locator. However, ranking this list is not straight forward, as the rank of a selector depends on the nature of task to be performed by the user and the information available in the DOM tree. Prior work by Keller et al. [17] provides metrics (such as 'universality') to rank the DOM element locator based on the information available in the DOM tree. However, to the best of our knowledge, none of the prior work has focussed on ranking the DOM element locator based on the nature of the task.

III. Methodology

Usage Model and Assumptions. We assume that the developer has a working web application, which is under development. This is required because we need to analyze the live DOM state and provide the developer with an interface to mark DOM elements as positive and negative examples. This is a reasonable assumption as the current state-of-art involves analyzing the DOM states manually, for which a live DOM state is needed.

Mathematical Model.: We define a mathematical model to convert the problem of synthesizing locator code into a constraint solving problem using the examples of DOM elements provided as inputs by the developer. Table III provides an overview of the mathematical model that we defined for this approach. The input required from the developer is in the form of positive and negative examples of DOM elements. The developer can also customize type of DOM element locator to be synthesized based on the nature of the task to be performed.

Approach Overview.: Our JavaScript code synthesis approach for selecting DOM elements is outlined in Figure 4, which consists of the following main phases: 1) Input collection, 2) Mathematical model generation, 3) Constraint solving, and 4)

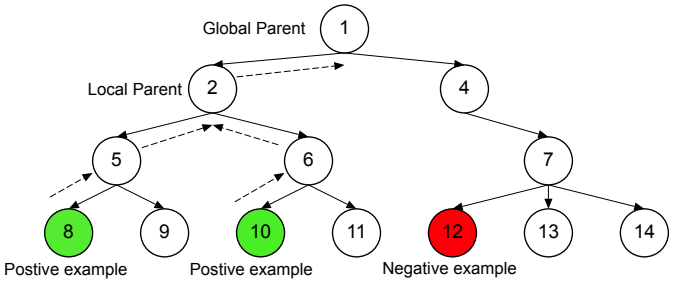

Fig. 5: Search scope within the DOM tree. Positive examples are shown in green, while negative examples are in red. 1) **Limited Scope:** only positive examples (Element 8,10) should be selected by the synthesized locator. 2) **Local Scope:** similar elements within the common parent (Element 2) of all positive examples are considered. 3) **Global Scope:** similar elements within the complete DOM tree are considered, root node of DOM tree (Element 1) acts as the root node for the search scope as well.

Algorithm 1: Generating CNF input from DOM Elements

Data: DOM Elements: Positive $\mathbb{P}$ and Negative $\mathbb{N}$ sets
Result: CNF Input for the SAT Solver

1 **foreach** $\mathbb{P}_i$ *in* $\mathbb{P}$ **do**
2 $\quad \mathbb{X}_i$ = generateAllPossibleDOMElementLocators($\mathbb{P}_i$);
3 $\quad [x_{i1}, x_{i2}, ...x_{in_i}]$ = filterUsingConfiguration($\mathbb{X}_i$);
4 $\quad DNF_i = x_{i1} \vee x_{i2} \vee ..... \vee x_{in_i}$;
5 $\quad i++$;
6 **end**
7 **foreach** $\mathbb{N}_j$ *in* $\mathbb{N}$ **do**
8 $\quad \mathbb{Y}_j$ = generateAllPossibleDOMElementLocators($\mathbb{N}_j$);
9 $\quad$ **foreach** y_j *in* $\mathbb{Y}_j$ **do**
10 $\quad\quad DNF_{i+k} = -y_j$;
11 $\quad\quad k++$;
12 $\quad$ **end**
13 **end**
14 $CNF = DNF_1 \wedge DNF_2 \wedge .... \wedge DNF_(i+k)$

Output generation. The *input collection* phase takes the DOM elements as input examples from the developer along with the set of configurations to guide the synthesis process. In the *Mathematical model generation* phase, we analyze each input example provided by the user and generate equations for the discovered constraints. The model is then passed as input to the SAT solver in the *constraint solving* phase. The solutions generated by the SAT solver are then converted into DOM element locators that can select the corresponding DOM elements in the *output phase*. In the next sections, we explain each phase of our approach in detail using the sample task described in Section II-A.

A. Phase 1: Input Collection

In this phase, the developer needs to provide positive examples ($\mathbb{P}$) of DOM elements that she wants to select, along with negative examples ($\mathbb{N}$) of elements that she wants to exclude. The intuition is that we look for similarities among the DOM elements provided as positive examples, and dissimilarities with the negative examples. The positive and negative examples together constitute the example set ($\mathbb{E}$). Note that $\mathbb{N}$ can be an empty set, but $\mathbb{P}$ must be non-empty. The developer also can define configuration parameters to guide the code synthesis process and limit the number of DOM element locators synthesized. They are as follows:

- **Depth of DOM element locator** ($\mathbb{D}$) Increasing the depth of DOM element locator can help prune unwanted DOM elements. However, it increases the dependency of the locator on the DOM, making the DOM element locator fragile to DOM manipulations. Therefore, the developer can limit the maximum depth of the DOM element locator.
- **Maximum execution time** ($\mathbb{T}$) The process of searching through the solutions may be time-consuming, especially when the number of possible solutions is large. Therefore, the developer can limit the time spent by LED to explore the possible solutions using the SAT solver.
- **Search scope** ($\mathbb{S}$) Search scope is used to limit the number of DOM elements to be selected. For example, if the developer wants to generate a DOM element locator to select a single element which is provided as a positive example, the developer can choose the *Limited* scope. However, if she wants to select multiple DOM elements, and the elements are within the same subtree, she can use *local* search scope. Or if the elements are across the entire DOM tree, she can use *global* search scope. Figure 5 provides an example of the different search scopes.

Sample task. Table III lists the input variables defined by the developer for the sample task described in Section II-A. To select elements 5-10 (Figure 2), assume the developer provides elements 5 and 6 as positive examples, and elements 4 and 11 as negative examples. We assume that all the configuration inputs are initialized with default values for simplicity's sake.

B. Phase 2: Mathematical Model Generation

In this phase, we convert the input DOM elements provided by the developer in the previous phase (Table III) as a single constraint in the *Conjunctive Normal Form (CNF)*, in order to input this to the SAT solver. Algorithm 1 provides an overview of steps required to convert a list of DOM elements into a CNF input for the SAT solver. For each DOM element in the set of positive elements (Lines 1-6), we first generate a list of all the possible DOM element locators to select that particular element (Line 2). The generated list is then filtered based on the configuration settings such as length of DOM element locator, selectors to be avoided, "must use" selectors, etc (Line 3). Next we generate a *Disjunctive Normal Form (DNF)* expression that combines all the available DOM element locators for each DOM element (Line 4). This is because we want to select at least one DOM element locator that can select the given DOM element. We repeat this process for each DOM element in the set of negative elements (Lines 7-13). Finally, we perform a conjunction of all the generated DNFs to form the CNF expression.

Sample task. Table IV shows the process of generating a CNF input for the SAT solver to solve the sample task described in the Section II-A. We first generate DOM element locators for the individual positive and negative elements (Row 1). The DOM element locators that do not satisfy the criteria specified by the developer (such as maximum depth) are then filtered from the DOM element locators for positive elements (Row 2). For each individual DOM element, we then generate a

TABLE III: Mathematical model for defining the input variables, and the values for the sample task.

Variable	Value	Definition	Sample Task
$\mathbb{E}$	$\{\mathbb{P}, \mathbb{N} \in DOM\ Elements\}$	Set of examples	$\{4, 5, 6, 11\}$
$\mathbb{P}$	$\{x \mid x \in Positive\ example\}$	Set of positive examples	$\{5, 6\}$
$\mathbb{N}$	$\{y \mid y \in Negative\ example\}$	Set of negative examples	$\{4, 11\}$
$\mathbb{D}$	1-20	Maximum depth of synthesized DOM element locator	4
$\mathbb{T}$	0-1000	Maximum execution time in seconds	10
$\mathbb{S}$	Limited search, Local search, Global search	Search scope for similar DOM elements	$Local\ search$
$\mathbb{U}$	user defined	List of DOM element property values that must be used in the synthesized locator	$null$
$\mathbb{A}$	user defined	List of DOM element property values that must be avoided in the synthesized locator	$null$
$\mathbb{O}$	user defined	Priority order various types of DOM element locators	$null$

TABLE IV: Mathematical model generation for the sample task

Row	DSL Code	Formula	$\mathbb{P}$ = Element 5,6	$\mathbb{N}$ = Element 4, 11
1	**DOM element locators ($\mathbb{X}_i$)**	$S(\mathbb{E}_i) = generateIndividualLocatorss(\mathbb{E}_i)$	body nav ul li, body nav li, body ul li, nav ul li, body li, nav li, ul li, li, body #globalheader #globalnav #gn-store, body nav ul #gn-store, body nav #globalnav #gn-store, body #globalheader #globalnav #gn-mac, body nav ul #gn-mac, body nav #globalnav #gn-mac, ...	body nav ul li, body nav li, body ul li, nav ul li, body li, nav li, ul li, li, body #globalheader #globalnav #gn-apple, body, nav ul #gn-apple, body nav #globalnav #gn-apple, body #globalheader #globalnav #gn-search, body, nav ul #gn-search, body nav #globalnav #gn-search, ...
2	**Filtered Locators F($\mathbb{X}_i$)**	$filter(S(\mathbb{P}_i), [\mathbb{D}, \mathbb{T}, \mathbb{U}, \mathbb{A}, \mathbb{S}, \mathbb{O}])$	body nav ul li, body nav ul #gn-store, body nav #globalnav #gn-store, body nav ul #gn-mac, body nav #globalnav #gn-mac, ...	...
3	**DNF(s)**	$\forall x \in \mathbb{P}\{(\cup F(S(x_i)))\}, \forall y \in \mathbb{N}\{\neg(\cup S(y_i))\}$	(body nav ul li ∨ body nav ul #gn-store ∨ body nav #globalnav #gn-store), (body nav ul li ∨ body nav ul #gn-mac ∨ body nav #globalnav #gn-mac), ...	¬(body nav ul li), ¬(body nav li), ¬(body ul li), ¬(nav ul li), ¬(body li), ¬(nav li), ¬(ul li), ¬(li), ¬(body #globalheader #globalnav #gn-apple), ¬(body), ¬(nav ul #gn-apple), ¬(body nav #globalnav #gn-apple), ¬(body #globalheader #globalnav #gn-search), ¬(body), ¬(nav ul #gn-search), ¬(body nav #globalnav #gn-search), ...
4	**CNF**	$\bigcap_{\forall x \in \mathbb{P}} (\cup F(S(x_i))) \cap \bigcap_{\forall y \in \mathbb{N}} \neg(\cup S(y_i))$	(body nav ul li ∨ body nav ul #gn-store ∨ body nav #globalnav #gn-store) ∧ (body nav ul li ∨ body nav ul #gn-mac ∨ body nav #globalnav #gn-mac) ∧ ¬(body nav ul li) ∧ ¬(body nav li) ∧ ¬(body ul li) ∧ ¬(nav ul li) ∧ ¬(body li) ∧ ¬(nav li) ∧ ¬(ul li) ∧ ¬(li) ∧ ¬(body #globalheader #globalnav #gn-apple) ∧ ¬(body) ∧ ¬(nav ul #gn-apple) ∧ ¬(body nav #globalnav #gn-apple) ∧ ¬(body #globalheader #globalnav #gn-search) ∧ ¬(body) ∧ ¬(nav ul #gn-search) ∧ ¬(body nav #globalnav #gn-search), ...	

DNF (Row 3) using the filtered DOM element locators. DNFs for individual examples are then combined to generate a CNF (Row 4). The generated CNF is then fed to the SAT solver in the next phase.

C. Phase 3: Constraint Solving

The constraints generated during the mathematical model generation are then passed to the SAT solver to provide a valid solution satisfying the constraints. Unfortunately, a generic SAT solver provides only a single solution to satisfy the given CNF [36]. However, to select the DOM element locator that is optimal (defined later), we need to find all the possible solutions for the given CNF and rank them. We use the technique defined in Zhao et al. [36] to recursively find all the possible solutions for the generated CNF. For each solution provided by the SAT solver, we create a blocking clause and append it to the previously used CNF. The CNF is then passed to the SAT solver to provide another possible solution. The process is repeated until all the possible solutions have been discovered, or the execution time provided as input by the developer is exceeded.

D. Phase 4: Output Generation

The output provided by the SAT solver is in the form a solution for the CNF input variables. We map these variables back to the DOM element locators to generate a list of DOM element locators that satisfy the constraints described in the mathematical model. The DOM element locators are then ranked based on various factors (discussed next) and the results are then presented to the user. We adopt and apply the *Synthesizer Driven Interaction* technique et al. [15] to avoid ambiguities, i.e., the system presents a series of input-output pairs to the developer, which she can mark as correct or incorrect. The algorithm is then re-run to provide the developer with an updated output.

Ranking DOM element locators. We rank the synthesized DOM element locators based on two main factors, i.e., 1) Information available in the DOM tree, and 2) Nature of the task to be performed by the JavaScript code. Prior work by Keller et al. [17] has defined two metrics i.e., *Universality*

TABLE V: Metrics proposed by Keller et al. [17] to rank the DOM element locators based on the information available in the DOM tree.

Metric	Definition
Universality	Ratio of number of `element` selectors versus the total number of selectors used within the DOM element locator. Increasing the number of `#id`, `.class` and other locators that utilize attribute specific information from DOM elements make the DOM element locator dependent on the DOM tree and should be avoided as much as possible.
Abstractness	Minimum of the number of restrictions posed by the DOM element locator on the DOM tree hierarchy and the universality score achieved by the DOM element locator. Increasing the depth of DOM element locator increases the number of restrictions on the DOM hierarchy and hence depth should be kept as less as possible.

and *Abstractness* (Table V) to rank the DOM element locators using the information available in the DOM tree. However, they do not take into account the task the user is attempting to perform. To capture the nature of task, we combine the information available in the configuration input specified by the user with the two metrics.

Sample task. For the sample task (Section II-A), the output consists of a list of DOM element locators, ranked based on the Universality and Abstractness of the synthesized DOM element locators. For the sake of simplicity, we only show the top 4 synthesized DOM element locators below.

1) `#gn-store, #gn-mac, #gn-ipod, #gn-iphone, #gn-ipad, #gn-itunes`
2) `ul #gn-store, ul #gn-mac, ul #gn-ipod, ul #gn-iphone, ul #gn-ipad, ul #gn-itunes`
3) `ul li #gn-store, ul li #gn-mac, ul li #gn-ipod, ul li #gn-iphone, ul li #gn-ipad, ul li #gn-itunes`
4) `nav ul #gn-store, nav ul #gn-mac, nav ul #gn-ipod, nav ul #gn-iphone, nav ul #gn-ipad, nav ul #gn-itunes`

IV. Implementation

We have implemented our approach in an open source tool called [21]. LED is built using JavaScript and can be installed as a Bookmarklet [32] within any web browser. Hence, it is not tied to a particular development environment. The source code for our tool is available to download, along with a video of the tool in operation [30]. Figure 6 presents a screenshot of LED.

In the **input phase**, the developer using LED can see the outlines of DOM elements by hovering over them, thus getting instant visual feedback on the element. She can select elements by dragging and dropping the DOM element into the input container as a positive or negative example. The developer can also configure the options for the synthesized DOM element locator based on the nature of the task to be performed. When finished, she needs to click on the *Generate Locator* button to initiate the DOM element locator code synthesis process of LED.

In the **mathematical model generation** phase, LED analyzes the DOM elements provided as examples by the developer. Using the DOM API methods, it then traverses the DOM tree for each example and captures the relevant information required to synthesize DOM element locators, and generate constraints for the SAT solver.

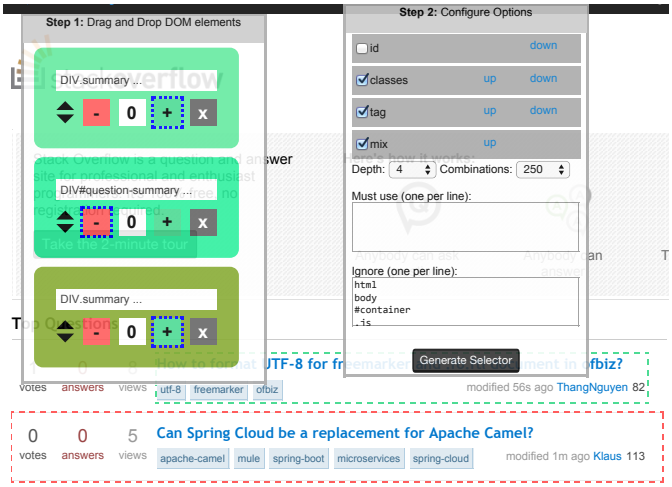

Fig. 6: Screenshot of DOM element locator generation using LED. Developers can mark the DOM elements as positive and negative examples, and configure options in order to generate the best possible DOM element locator.

In the **constraint solving** phase, LED initiates the SAT solver using the constraints defined in the previous phase. We use the Minisat solver [31].

In the **output phase**, the results obtained by SAT solver are presented to the developer in the form of ranked DOM element locators. The output phase is tailored towards handling the possible ambiguities present in the input provided by the developer. When the developer hovers over a locator, LED provides live feedback by highlighting the DOM elements that are selected by that locator, and she can mark those as correct and incorrect.

The **current implementation** of LED supports synthesizing DOM element locators that belong to CSS1 [6] and CSS2 [7] category. DOM element locators that belong to CSS3 [16] category can also be synthesized using a similar approach. However, synthesizing such locators require additional information from DOM as well as from developer which we currently discard. Extending our approach to synthesize DOM element locators for the CSS3 category is a direction for future work.

V. Evaluation

In this section, we first define the research questions to assess the overall approach, and then define a methodology to answer each of the research questions.

A. Goals and Research Questions

To evaluate how effectively LED can synthesize input parameters for DOM API functions, we answer the following research questions in our study.

RQ1: Do developers access multiple DOM elements with a single DOM element locator? How complex are the DOM element locators used by web developers within JavaScript code?

RQ2: What types of DOM element locators are predominantly used by web developers? What percentage of those DOM element locators are supported by LED?

RQ3: How accurate are the DOM element locators synthesized by LED, with only positive examples, and with both positive and negative examples?

RQ4: What is the performance overhead incurred by LED?

B. Methodology

RQ1: Complexity. We use the Phormer [27], Gallery3 [11], and Wordpress [33] web applications to measure the complexity of DOM element locators used by web developers. These are large and popularly used web applications, which have also been used in our prior work [4].

For each web application, we intercept the calls to DOM API methods and recorded the DOM element locator used by the web developers. We then measure the number of DOM elements selected by the DOM element locator in the web application. Further, a DOM element locator can be formed using a combination of atomic DOM element locators such as `#id`, `.class`, and `tag`. Therefore, we also measure the number of atomic DOM element locators used within a single DOM element locator. This gives an indication of the complexity of the DOM element locator.

RQ2: Coverage. As discussed in Section IV, LED supports only a limited set of DOM element locators. We investigate to what degree these locators are used in practice. One way to assess this is by doing an exhaustive study of DOM element locator usage in real web applications. However, due to time restrictions, we restrict ourselves to DOM element locator usage in the landing pages of the Alexa top 200 websites. Since the JavaScript code used in production level websites is often minified and obfuscated, we chose to analyze DOM element locators used within their stylesheets for this study. For each web application, we extract the style sheets used by it. For each stylesheet, we measure the total number of DOM element locators, and the number of DOM element locators that are supported by the current version of LED. We chose to analyze stylesheets rather than JavaScript code as JavaScript code in production web applications is often obfuscated and minimized, making it difficult to parse and analyze. Note that LED is a development phase tool, and hence does not have to deal with minified or obfuscated code.

RQ3: Accuracy. We assess whether the DOM element locators synthesized by LED match the ones written by the programmer. This assumes that the DOM element locator written by the developer in the JavaScript code is correct. We use the same three web applications as used in the first research question.

For each application, similar to RQ1, we intercept the calls to the DOM API methods within the JavaScript code, and record the DOM elements selected by the method. As DOM API methods are implemented using DOM element locators, this is equivalent to capturing the DOM element locators written by the developer. We then use LED to synthesize a DOM element locator using the given DOM elements as positive examples. We also consider the effect of including negative examples from the remaining elements. A DOM element locator is considered adequate if it selects the DOM elements that were also selected using the original DOM element locator used by the developer.

To measure the accuracy of our proposed technique we follow the following procedure for each application:

1) Crawl / Interact with the web application.
2) Intercept calls to DOM API methods and record the DOM objects returned by the method in the JavaScript code.
3) Pass the DOM elements as positive examples to the system, both with and without negative examples from DOM.
4) Analyze the synthesized DOM element locator to measure the recall and precision of LED.

RQ4: Performance. For assessing the performance of our code synthesis technique, we analyze the time taken by LED to synthesize the list of DOM element locators for the above three web applications. We then report the average time taken to synthesize DOM element locators for each of the three web applications with respect to different search configurations.

TABLE VI: Complexity of DOM element locators

No. of selected DOM elements	Percentage of use cases	No. of atomic DOM element locators	Percentage of use cases
1	78.17%	1	65.85%
2 - 5	11.97%	2	21.83%
6 - 10	1.41%	3	2.46%
11 - 100	8.10%	4	9.51%
$>$ 100	0.35%	$>$5	0.35%

C. DOM element locator Complexity

Table VI provides an overview of the complexity of DOM element locators used by web developers to select DOM elements within the JavaScript code. As shown in the table, about 22% of DOM element locators written by web developers target multiple DOM elements, with about 10% of the DOM element locators target more than 5 DOM elements. Therefore, when writing DOM element locators, the developer needs to carefully analyze and abstract the common information among the target DOM elements. This is where a tool like LED really helps, as it automatically finds this information for the selected DOM elements.

Also, we can see that about 35% of the DOM element locators are formed by combining multiple atomic DOM element locators. This is typically done to avoid certain DOM elements that do not satisfy the locator criteria. Therefore, writing DOM element locators to target a specific set of DOM elements within large web application is a non-trivial task for the developer. Again, LED can help here as it allows developers to specify both positive and negative examples easily.

***Finding 1**: 22% of the total DOM element locators written by web developers are used to select multiple DOM elements, and 35% of the DOM element locators are a combination of multiple atomic DOM element locators.*

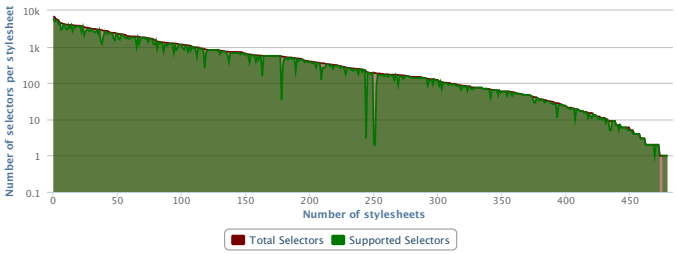

Fig. 7: DOM element locator coverage using LED for 486 stylesheets in top 200 websites

TABLE VII: Categorization of DOM element locators synthesized using LED

Category	Type	Definition
Adequate	Essential	DOM element locator that selects all the elements intended by the user, and nothing else.
	Auxillary	DOM element locator selects the DOM elements intended by the user and additional DOM elements.
Inadequate		DOM element locator that does not select all the elements intended by the user.

D. DOM element locator coverage

We analyzed the DOM element locators used by web developers in the Alexa top 200 applications. Figure 7 presents the results of our analysis, which includes a total number of 486 stylesheets, with an average of 650 DOM element locators per stylesheet. The area under the red curve represents the total number of DOM element locators used by the web developers, and the area under the green curve represents the total number of DOM element locators that are supported by the current implementation of LED, and could have been synthesized using our approach (in theory). As shown in the results, the majority of the DOM element locators used by web developers is supported by LED. With an exception of few stylesheets, more than 90% of the DOM element locators in each stylesheet are supported by LED. One prominent locator that is not supported by LED is the `[attribute=value]` DOM element locator. To support this locator, we need to capture additional information from the DOM tree that we currently discard. However, the underlying approach remains the same.

***Finding 2**: LED supports 86% of the existing DOM element locators used by web developers of the most popular web applications.*

E. Accuracy

We use the commonly used metrics i.e., recall and precision to measure the efficacy of our proposed system. We categorize the synthesized DOM element locators into two major categories i.e., adequate and inadequate. In general, a DOM element locator is only considered adequate if it can at least select the elements which the developer intended to select. If a DOM element locator cannot select all the DOM elements that were selected using the intercepted DOM element locator, it is marked as inadequate. Table VII provides a brief overview of the categorization of synthesized DOM element locators. Below we discuss how we define precision and recall metrics for our analysis.

TABLE VIII: Recall and Precision achieved by LED using various inputs with $\phi = 5$

Trial	No. of positive examples	No. of negative examples	Recall	Precision
0	$<= \phi$	0	98.21%	48.03%
1	$> \phi$	0	100.00%	47.85%
2	$<= \phi$	ϕ (random)	98.05%	62.82%
3	$> \phi$	ϕ (random)	100.00%	62.98%
4	$<= \phi$	ϕ (relevant)	98.05%	91.84%
5	$> \phi$	ϕ (relevant)	100.00%	92.05%

Recall. Recall is a measure of the extent to which LED can synthesize DOM element locators that can select the elements intended by the user. We measure recall as the ratio of adequate DOM element locators synthesized by LED to the total number of DOM element locators synthesized by LED.

Precision. Precision is a measure of an extent to which LED can synthesize DOM element locators that can select the exact elements intended by the user. We measure precision as a ratio of essential DOM element locators versus the total number of adequate (essential + auxiliary) DOM element locators synthesized by LED.

Table VIII provides an overview of the accuracy achieved by LED. As the number of examples provided by the developer can influence the DOM element locators produced by our tool, we measure precision and recall for different scenarios. We chose a threshold of $\phi = 5$ to determine the number of positive and negative examples. The choice of $\phi = 5$ was based on the balance between the number of examples required to synthesize meaningful code versus the effort required by the developer. Further, we found in RQ1 that 90% of the DOM element locators in these web application select 5 elements or fewer, so this should be sufficient for our purposes.

The number of positive examples has 2 values $\{<= \ \phi, > \ \phi\}$ and negative examples has 3 values $\{None, \phi(Random), \phi(Relevant)\}$. The negative examples were chosen using the different methods described below:

None(0) No negative examples were chosen.

Random The negative examples were chosen randomly from the set of all DOM elements that are not included in the original DOM element locator provided by the developer.

Relevant The negative examples were chosen from the set of DOM elements of the same type as the elements selected by the original DOM element locator. For example, if an intercepted DOM element locator selects a subset of anchor tag elements ($< a >$), the negative examples were chosen from the remaining set of the anchor tag elements within the DOM.

As seen from the results, even with a small number of positive examples i.e., $<= \phi$, LED achieves high recall (98%). Since, we used the same set of web applications as RQ1, according to Finding 1 , 78% of the DOM element locators that we analyzed were used to target a single DOM element. Hence, even with a single positive example the recall achieved by LED is very high, demonstrating that the amount of effort required by the developer is minimal.

When the positive examples set is expanded to the entire set of DOM elements chosen by the developer's DOM element

TABLE IX: Performance overhead caused by LED.

Search Scope	Average time per application (seconds)			
	Phormer	**Gallery3**	**WordPress**	**Average**
Limited search	0.05	0.08	0.46	0.20
Local search	0.06	0.10	0.48	0.21
Global search	0.07	0.11	0.49	0.22

locator, i.e., $>= \phi$, the recall increases to 100%. This means that any DOM element locator synthesized by LED will select at least the DOM elements that the developer intended to select.

The precision achieved by LED without any negative examples (0) is only around 48%. However, when even a small number of random negative examples are provided i.e., $\phi(random)$, the precision increases to 62%. The addition of negative examples assists our approach in excluding the DOM element locators that select more than the required DOM elements. Further, when the negative examples were chosen to be "relevant" ones, i.e., $\phi(relevant)$, the overall precision increases to 92%, showing that it is important to give good negative examples to get high precision. Note that the precision results do not vary much with the number of positive examples provided, however, in all three cases.

Finding 3*: LED can synthesize DOM element locators with a 98% recall and 92% precision, with as low as 5 positive and 5 relevant negative examples.*

F. Performance Overhead

The time taken by LED to synthesize DOM element locators depends on 1) the number of examples provided as input by the developer, 2) size of the DOM tree, and 3) the configuration input provided by the user. To measure the effect of different DOM sizes of DOM tree, we report the time taken to synthesize DOM element locators for the three different web applications in our case study. As discussed in Section III-A, there are different input configurations that the developer can specify before synthesizing the DOM element locators. One such configuration described in Section III-A is the search scope of the target DOM elements. The developer can limit the search space within the target DOM tree (i.e., local search), therefore limiting the number of available DOM elements for analysis. The more the number of DOM elements to analyze, higher is the time taken to synthesize DOM element locators. Therefore, when synthesizing DOM element locators we consider three different scenarios (i.e., only positive examples, local search, and global search) for each application. The overall approach to measure the time taken by LED is similar to Section V-E. However, this time we only consider 5 positive examples, and 5 negative examples.

Table IX shows the average time taken to synthesize DOM element locators for each of the three web applications. As we can see from the results, the time taken to synthesize DOM element locators varies with each application. However, the change in time with respect to search scope is not significant. The average time is 0.22 seconds while the maximum time is 0.49 seconds, showing that LED is fast, and can be deployed interactively during development.

Finding 4*: LED can synthesize DOM element locators with an average time of 0.22 seconds, with a worst case of maximum 0.5 seconds, for the selected web applications.*

VI. Discussion

In this section, we discuss the limitations of LED, and the threats to validity of our evaluation.

A. Limitations

Single DOM state. LED supports only a single DOM state for synthesizing a DOM element locator. However, the developer may want to generate a single DOM element locator for elements that span across multiple DOM states. The current version of LED does not support this feature. In our future work, we plan to extend LED to support DOM element selection from multiple DOM states as well. The underlying approach of analyzing the DOM elements and generating constraints for the SAT solver will remain the same. However, we might need to optimize the performance of our algorithm to scale for inputs from multiple DOM states.

Limited support for DOM element locators. DOM element locators can be divided into three categories, based on the version of CSS in which they were defined, i.e., CSS1, CSS2, and CSS3. A complete list of DOM element locators along with their category can be found on the w3schools website [8]. Currently LED supports synthesis of DOM element locators that belong to first two categories i.e., CSS1 and CSS2. Locators that belong to the CSS3 category (e.g., attribute-value locator), require extra information (e.g., element name, element value, etc.) from the DOM, that we discard currently. However, our results demonstrate that LED can support 86% of the DOM element locators used by web developers.

DOM element visibility. The developer needs to select the DOM elements as examples to initiate the code synthesis process. For the developer to select the DOM elements, the corresponding DOM elements need to be visible within the web application. However, it is possible that the target DOM elements are not visible within the live DOM state. For example, a warning message to the user may not be visible. To enable the developer to select the hidden DOM elements, we will need to modify attributes of hidden DOM elements. This may adversely affect the layout and functionality of the web application. The user can add information regarding these hidden-elements directly to the mathematical model. However, in future work, we plan to extend our approach to analyze the hidden DOM state and allow users to provide hidden DOM elements as input examples.

B. Threats to validity

Internal Threat. In our evaluation of the accuracy LED, we considered three kinds of negative examples, namely none, random and relevant. The relevant examples were chosen based on the similarity among the types of DOM elements

within the DOM tree. However, this process is subjective and may depend on the user. This may affect the overall quality of negative examples provided by the developer, which in turn will affect the overall precision achieved by the tool. We have attempted to mitigate this threat by considering similar types of elements as the ones being chosen as relevant negative examples.

Another internal threat to our validity is that we chose a threshold of $\phi = 5$ for the number of input examples. This threshold was chosen based on what we thought was a reasonable between user effort and sufficient information to synthesize locators. We did not observe too much difference when we increased the threshold. However, we did see a decrease in accuracy when we decreased the threshold.

External Threat. To evaluate the DOM element locator coverage of LED, we studied the top 200 Alexa websites. It is possible that coverage results may differ if we consider more applications. This is an external threat. However, these top 200 web applications are the most popular web applications, and as such, are likely to be representative of other web applications. Another external threat to the validity of our results is that we have considered limited number of open source web applications for our evaluation. However, the chosen web applications were commonly used open source web applications.

Reproducibility of results. Because web applications evolve over time, the DOM element locators used today may not be prevalent in the future. This may affect the selector coverage and accuracy results for LED. Therefore, we have made our analyzed dataset publicly available for download [20].

VII. Related Work

Closely related to our work is the problem of XPath generation, which is one of the method for locating DOM elements within the DOM tree. Tools such as Firepath [28], XPath Helper [12], and Xpath Checker [34] also generate XPath expressions for DOM elements. There has been an extensive body of work on generating DOM element locators for test cases in web applications [1], [2], [5], [29].In very recent work, Leotta et al. [22] design a voting algorithm to select the most robust DOM element locator generated using existing techniques, thereby increasing the robustness of the generated locators. However, all of these techniques are restricted to generating locators for single DOM elements unlike our work, which can generate locators for multiple DOM elements. Further, none of these techniques generate DOM element locators from positive and negative examples of DOM elements, like our technique does.

Our earlier work [4] provided a technique to assist the web developers in writing JavaScript code by providing DOM aware JavaScript code completion. However, the developer still needed to write at least part of the DOM element locator expression to invoke the code completion technique. In this work, we present a technique to synthesize the CSS selectors based on examples, thereby minimizing the time and effort required by the developer.

Our technique is an example of Live Programming, and in particular Programming By Example (PBE). There has been a significant amount of work performed in this area [19], [24], [25], [13], [18], [10]. Most of these focus on learning string or data manipulation actions, while we focus on learning CSS selectors for selecting DOM elements in web applications.

Another class of live programming techniques are Programming By Demonstration (PBD), which is also related to our technique. The main difference is that instead of using the input and output states, the transformations are inferred based on the trace of actions performed by the user. For example, Yessenov et al. [35] presents an approach where the user can denote different parts of the text with different colors, and the hierarchical structures present in the text document is automatically learned. This is similar to synthesizing CSS selectors for DOM elements. The main difference between this work and ours is that in our case, the JavaScript code that we synthesize spans multiple domains, namely DOM and JavaScript and is hence much more challenging to synthesize.

VIII. Conclusion and Future Work

In this paper, we introduced a DOM element locator synthesis technique based on positive and negative examples of DOM elements provided as input by the developer. Our approach is based on generating constraints using DOM elements given as input in the form of a mathematical model and then utilizing constraint solvers to find possible solutions. We have implemented our approach in an open source tool called LED. The results of our evaluation show that LED is compatible with over 86% of the existing DOM element locators used by the web developers, and can synthesize DOM element locators with a 98% recall and 92% precision, even with just a few positive and negative examples. The maximum time taken by LED to synthesize DOM element locators is less than 0.5 seconds, making it practical for interactive synthesis.

We plan to extend this paper in a number of ways. First, we plan to extend our approach to cover the CSS3 locators that require integrating additional information from the user as well as the DOM state. We also plan to extend our approach to support XPath generation for multiple DOM elements. Further, we plan to extend the evaluation of our work and conduct user studies to analyze how effectively can LED assist the web developers in writing JavaScript code that interacts with DOM. Finally, we plan to extend the implementation of our tool to support multiple DOM states.

IX. Acknowledgements

We thank the anonymous reviewers of the ASE'15 conference for their feedback that helped improved the paper. This work was supported in part by an NSERC Strategic Grant, a MITACS fellowship and a research gift from Intel Corporation.

REFERENCES

[1] M. Abe, S. D. DeWitt, M. Hori, and B. B. Topol. Generating and utilizing robust XPath expressions, 2006. US Patent 7,086,042.

[2] M. Abe, S. D. DeWitt, M. Hori, and B. B. Topol. Selectable methods for generating robust xpath expressions, 2007. US Patent 7,213,200.

[3] Apple (canada). http://www.apple.com/ca/. Accessed: 2014-08-26.

[4] K. Bajaj, K. Pattabiraman, and A. Mesbah. Dompletion: DOM-aware JavaScript code completion. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 43–54. ACM, 2014.

[5] C. Boehm, K. S. Devlin, T. Peterson, and H. Singh. System and method for generating an xpath expression, 2006. US Patent App. 11/368,292.

[6] Cascading style sheets, level 1. http://www.w3.org/TR/REC-CSS1/. Accessed: 2015-08-09.

[7] Cascading style sheets, level 2. http://www.w3.org/TR/CSS2/. Accessed: 2015-08-09.

[8] CSS selectors reference. http://www.w3schools.com/cssref/css_selectors.asp. Accessed: 2015-08-09.

[9] A. Cypher and D. C. Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

[10] Y. Fujishima. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of the 3rd international conference on Intelligent user interfaces*, pages 101–108. ACM, 1998.

[11] Gallery 3 begins | gallery. http://galleryproject.org/gallery_3_begins. Accessed: 2014-04-23.

[12] Google/xpaf. https://github.com/google/xpaf. Accessed: 2015-08-09.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, pages 317–330. ACM, 2011.

[14] S. Gulwani. Synthesis from examples. In *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 2012.

[15] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.

[16] Introduction to CSS3. http://www.w3.org/TR/2001/WD-css3-roadmap-20010523/. Accessed: 2015-08-09.

[17] M. Keller and M. Nussbaumer. Css code quality: A metric for abstractness; or why humans beat machines in CSS coding. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 116–121. IEEE, 2010.

[18] J. Landauer and M. Hirakawa. Visual AWK: a model for text processing by demonstration. In *Visual Languages, IEEE Symposium on*, pages 267–267. IEEE, 1995.

[19] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Learning repetitive text-editing procedures with SMARTedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, pages 209–226, 2001.

[20] LED dataset. http://ece.ubc.ca/~know/led/data.zip. Accessed: 2015-08-09.

[21] LED demo. http://ece.ubc.ca/~kbajaj/led.html. Accessed: 2015-08-09.

[22] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[23] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[24] R. C. Miller, B. A. Myers, et al. Lightweight structured text processing. In *USENIX Annual Technical Conference, General Track*, pages 131–144. USENIX, 1999.

[25] B. A. Myers. Tourmaline: Text formatting by demonstration, 1993.

[26] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64. IEEE Computer Society, 2013.

[27] Phormer, the PHP without MySQL photogallery manager. http://p.horm.org/er/. Accessed: 2014-04-23.

[28] Pierretholence/firepath. https://code.google.com/p/firepath/. Accessed: 2015-08-09.

[29] F. Ricca, M. Leotta, A. Stocco, D. Clerissi, and P. Tonella. Web testware evolution. In *Web Systems Evolution (WSE), 2013 15th IEEE International Symposium on*, pages 39–44. IEEE, 2013.

[30] Saltlab/LED. https://github.com/saltlab/led. Accessed: 2015-08-09.

[31] N. Sorensson and N. Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

[32] Use bookmarklets to quickly perform common web page tasks | firefox help. https://support.mozilla.org/en-US/kb/bookmarklets-perform-common-web-page-tasks. Accessed: 2015-08-09.

[33] Wordpress: Blog tool, publishing platform, and CMS. https://wordpress.org. Accessed: 2014-04-23.

[34] Xpathchecker - an interactive editor for xpath expressions - firefox extension - google project hosting. https://code.google.com/p/xpathchecker/. Accessed: 2015-08-09.

[35] K. Yessenov, S. Tulsiani, A. Menon, R. C. Miller, S. Gulwani, B. Lampson, and A. Kalai. A colorful approach to text processing by example. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 495–504. ACM, 2013.

[36] W. Zhao and W. Wu. Asig: An all-solution sat solver for cnf formulas. In *Computer-Aided Design and Computer Graphics, 2009. CAD/Graphics '09. 11th IEEE International Conference on*, pages 508–513. IEEE, 2009.