WebChecker: An Epsilon Validation Language Plugin

Tebin M. Raouf
Computer Science Dept.
College of Staten Island, CUNY
Staten Island, NY 11314, U.S.A.
tebin.raouf@cix.csi.cuny.edu

Dimitrios S. Kolovos
Computer Science Dept.
University of York
Deramore Lane, York, YO10 5GH, UK.
dimitris.kolovos@york.ac.uk

Xiaowen Zhang
Computer Science Dept.
College of Staten Island, CUNY
Staten Island, NY 11314, U.S.A.
xiaowen.zhang@csi.cuny.edu

***Abstract*—WebChecker is an Epsilon Validation Language (EVL) plugin to validate static and dynamic Hyptertext Markup Language (HTML) pages that use frameworks such as Bootstrap by writing configurable EVL constraints. WebChecker is used for enforcing implicit rules of HTML and Cascading Style Sheet (CSS) frameworks. To show the effectiveness of the plugin, we choose Bootstrap, the world's most popular HTML, CSS, and JavaScript framework. The plugin comes with a list of EVL constraints for checking web pages built with Bootstrap. To support our argument, we present a concrete example with two solutions for enforcing implicit rules.**

Keywords-Lint, Web Lint, EVL, Epsilon, HTML, Java, Bootstrap.

I. Introduction

In the current state of web development, there are many HTML and CSS frameworks that come with implicit rules. Such frameworks include Bootstrap [1], Materialize [2], Foundation [3], and uikit [4]. These rules are developed for better use of such frameworks and best practices. Therefore, not following such rules impact appearances and framework features. Most of these frameworks are built to be responsive on different screen sizes such as large screens of personal computers and small screens of smart devices. To provide a consistent look and feel on all devices, it is important to follow such implicit rules.

The Accessible Rich Internet Applications (ARIA) [5] specification introduces a handful number of rules for web pages in order to make them accessible for users with disabilities through assistive technologies such as screen readers. Such assistive technologies require web pages have certain attributes defined or enabled on HTML elements. Bootstrap, for example, provide default classes and ARIA attribute values throughout the framework for such technologies. Therefore, conforming to such rules make web pages available to a greater audience.

However, the caveat is to read frameworks' documentations carefully to capture such constraints and rules. This could be quite a tedious work and expensive. WebChecker is built to capture such rules so that users of these framework can easily check their web pages for the conformance of such rules. While WebChecker is built with HTML frameworks in mind, it could be easily used for web pages built without such frameworks. However, in this paper, we focus on the Bootstrap framework. Furthermore, WebChecker is able to check static and dynamic pages, which are generated by scripting languages such as PHP and JavaScript.

The rest of the paper is structured as follows: Section II explains the problem and the current state of art solution and its limitations that motivated this project. In Section III, we explain the WebChecker plugin and we show our solution for the problem. In Section IV, we present a few sample implicit rules captured for the Bootstrap framework. We compare the current solution and our solution of the problem and its improvements in Section V. In Section VI, we discuss related work. Lastly, Section VII concludes the paper and outlines the future of the plugin and future work to enhance the plugin.

II. Background and Motivation

Frameworks such as Bootstrap come with a handful number of implicit rules. While developing this project, we captured a couple of dozen rules for the Bootstrap framework. These rules should be followed in order to use the framework properly. These rules are created by the developers of such frameworks and explained in the framework's documentation in a natural language. Users of such frameworks should carefully read the documentations in order to understand how to use the framework. This process can be quite long and tedious, expensive, and prone to errors. Given a set of rules, currently there is not a straightforward process to check if they are enforced against the HTML pages. At least, there is not an easy way that requires minimum effort and very little code. After collecting the rules, the current solution flow is: 1. Read and encode the HTML page. 2. Translate each rule into a method. 3. Check the rule against the HTML page. 4. Repeat 1-3 for every rule. For example, in Bootstrap, any content should be under a * * element with class *col*, which should be under a * * element with class *row*, which should be under another * * element with class *container*. The HTML code sample is at Listing 1.

```
1<!— sample.html —>
2<div class="container">
3  <div class="row">
4    <div class="col">
5      Column One
6    </div>
7    <div class="col">
```

```
8       Column Two
9     </div>
10    <div class="col">
11      Column Three
12    </div>
13  </div>
14</div>
```

Listing 1: Bootstrap Grid Example

To enforce this rule, the current state of the art solution is given in Listing 2. Since the plugin is written in Java, we choose it as our programming language of choice to implement this solution. However, any programming languages can be used. Some of which might take more or less coding. Listing 2 shows, for enforcing the rule in Listing 1, the code can become lengthy and unreadable, which makes the logic confusing to reason about. It is also difficult to reuse and maintain this code since each rule has its own requirements and structure. Depending on the rules, it is likely to have more nested *if statements*. Furthermore, the solution is not configurable. That is, it is difficult to make changes without breaking the code logic. This approach requires framework developers and users to spend quite some time to write similar code as Listing 2 to enforce rules and constraints, and write documentations for such code.

The above difficulties have motivated us to implement a plugin, WebChecker, for the Epsilon Validation Language (EVL) [6]. This plugin is designed for enforcing rules and constraints for any frameworks such as Bootstrap. With WebChecker, 1) a static or dynamic HTML file, 2) an HTML web page through its URL, and 3) a specific section of an HTML page can be checked for conformance. This plugin provides EVL constraint reusability across multiple projects and improves readability. Framework users and developers are able to focus on the rules and constraints. The next section explains WebChecker in more details.

```
1  File input = new File("files/bootstrap/newCheck.
       html");
2  try {
3   Document doc = Jsoup.parse(input, "UTF-8");
4   Elements elements = doc.getElementsByTag("div")
        ;
5   for (Element element : elements) {
6    if (element.hasClass("col-sm-4")) {
7     if (!element.parent().hasClass("row")) {
8      System.out.println("A div element with class
           col should have a parent element with
          class row");
9     } else {
10     if (!element.parent().parent().hasClass("
           container")) {
11      System.out.println("A div element with
           class col should have a parent element
           with class row, which has a parent with
            class container.");
12     }
13    }
14   }
15  }
16  } catch (IOException e) {
17   e.printStackTrace();
18  }
```

Listing 2: Enforcing Bootstrap Grid Rule

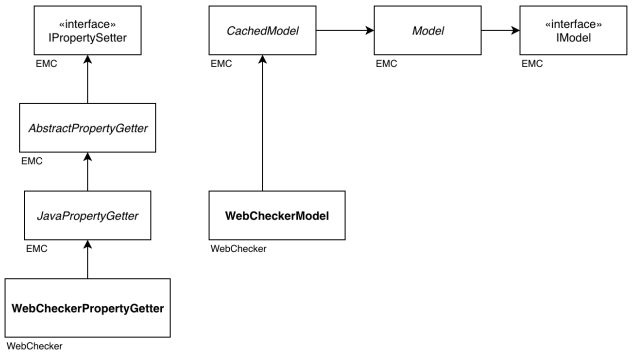

Figure 1: WebChecker and Partial EMC Diagram

III. WebChecker Plugin

Epsilon Validation Language (EVL) [6] is one of the languages of Epsilon [7], which is an Eclipse project that provides languages for model managements such as model validation, model transformation, code generation, pattern matching, model merging and etc. In the context of WebChecker, a model is an HTML page or a section of the page. WebChecker implements Epsilon Model Connectivity (EMC) layer that provides an interface, IModel, to capture the HTML model and EVL constraint file. The two main classes implemented by the plugin are *WebCheckerModel* and *WebCheckerPropertyGetter* while the other classes are provided by Epsilon. Figure 1 shows this relation. Furthermore, EVL users can use the features of WebChecker without setting up a new development workflow or installing new software.

A. WebChecker

The WebChecker flow, after collecting the constraints, is: 1) Write an EVL constraint. 2) Choose the HTML source (i.e. an HTML file, URL, or section of HTML). These two steps are modular and readable. Listing 4 is an example of EVL file, which shows the solution for Listing 1.

WebChecker follows the *separation of concern*[1] principle with writing minimum code. In particular, there is an EVL source file and an HTML source, which could be an HTML file content or its URL. The EVL file that contains the constraints such as Listing 4 is run against the HTML source. Listing 3 shows this connection. Line 10 and 11 of Listing 3 set the HTML and EVL source respectively. Moreover, line 15 of Listing 3 shows how to handle errors returned from the EVL constraints. This way the framework user and developer can only focus on constraints that do not satisfy.

```
1  //sample.java
2  //Step 1: Get the source to be validated
3  String html = "files/bootstrap/newCheck.html";
4
```

[1]According to Wikipedia *separation of concern* is as a design principle for separating a computer program into distinct sections.

```
5  //Step 2: Write your validation using Epsilon
       Validation Language
6  String evl = "files/bootstrap/newCheck.evl";
7
8  //Step 3: Check the validation against the html
       in step 1
9  WebChecker checker = new WebChecker();
10 checker.setSource(html);
11 checker.setValidation(evl);
12 checker.check();
13
14 //Step 4: Check the result
15 List<String> errors = checker.errors();
```

Listing 3: Checking an EVL file against an HTML source file

B. WebChecker EVL Structure

The WebChecker EVL structure follows the same EVL structure [6]. However, here we only explain the current syntax used by WebChecker.

```
1 //sample.evl
2 context t_div {
3     constraint DivWithColHasRowParent {
4         guard : self.class.includes("col-*")
5         check : self.parent.hasClass("row") and
              self.parent.is("div")
6         message : "A <div> element with class col
              should have a parent <div> element
              with class row."
7     }
8     constraint DivWithRowHasContainerParent {
9         guard : self.class.includes("row")
10        check : self.parent.hasClass("container")
              and self.parent.is("div")
11        message : "A <div> element with class col
              should have a parent <div> element
              with class row."
12    }
13 }
```

Listing 4: Enforcing Bootstrap Grid Rule by Using WebChecker

- *Context*: an EVL file could have one or more contexts. To capture an HTML element, a context should start with *t_*, for *type*, followed by the name of the HTML element such as *t_div*, *t_picture*, *t_section*, *t_button*, and etc.
- *Constraint*: each *Context* could have one or more constraints. A *constraint* has a name that is used to differentiate each constraint from each other. While the constraint name does not matter, we suggest having a name that identifies the rule. Each *constraint* has three blocks *guard*, *check*, and *message*. Optionally, an EVL constraint could have a *fix* section, where the model is fixed if the constraint does not satisfy. Currently, WebChecker does not utilize this feature. Section VI explains future enhancements to WebChecker where the *fix* block is explained.
- *guard*: this block captures a specific section of the context for checking. *guard* must return a boolean value. If true, the *check* block is executed. The *guard* block accepts compound boolean statements by using *and* and *or*.
- *check*: after the *guard* block is satisfied, this block is executed to *check* if the constraint is satisfied. Expressions in this block must return a boolean value. If the expression returns true, the constraint is satisfied and hence the *message* block is not executed. Similarly as the *guard* block, this block accepts compound boolean statements.
- *message*: this block returns a message if the constraint is not satisfied. WebChecker provides a convenient method to capture all unsatified constraints' messages shown in Line 15 of Listing 4.

IV. Sample Bootstrap Implicit Rules

In this section, we show and explain a few implicit rules captured for the Bootstrap framework. The following are randomly chosen and more rules can be accessed from the plugin repository [2] For brevity, we do not show the actual EVL syntax.

- *ScreenReaderButton*: this constraint captures buttons with class *close* and validates if the *<button>* element has the *aria-label* attribute defined for assistive technologies.
- *AlertLinkInDivAlert*: this constraint captures *<a>* elements with class *alert-link* and validates if the parent element includes *alert* and *alert-** classes. The asterisk (*) is a wild card such as *alert-success* or *alert-danger* and etc.
- *BtnGroupToggle*: this constraint captures * * elements with class *btn-group-toggle* and checks if its *data-toggle* attribute is defined and its value is equal to *buttons*.

V. Improvements over Current Solution

As shown from the example above, WebChecker is built to be easy, modular, readable, and configurable. These features address and solve most of the challenges developers face while using a program analyzer as studied by Microsoft researches [8]. WebChecker, specifically, offers custom warning messages, platform independency, static and dynamic page analysis, and easy integration into development workflow. Importantly, it is left to the developer to check HTML pages against any set of EVL constraints, which makes WebChecker stand out among other validators that have default un-configurable rules.

Furthermore, WebChecker provides an abstract class, *WebChecker*, to interact with the underlying model. While users can (should) use the *WebChecker* class, they are not restricted to use the *WebCheckerModel* class, which implements CachedModel of EMC. While *WebCheckerModel* requires background knowledge on Epsilon, it could be

[2] https://github.com/tebinraouf/webchecker

very powerful and configurable. The WebChecker abstract class allows a user to focus on the constraints rather than writing code, which saves time and resources. WebChecker modularity allows it to be reusable and extensible. If new framework rules are in place, WebChecker can be modified or extended to support such rules.

Code written like Listing 2 is prone to errors and confusing, which makes the code unreadable to none technical stakeholders such as managers and web designers. Also, as stated above the code in Listing 2 is written in Java. It is likely the developer uses a language that their workflow uses for validation such as JavaScript, PHP, C#, Python and etc. This makes it difficult to use the code in other development environment that uses different programming languages and technology stack. In contrast, WebChecker's EVL constraints are easy to reason about and independent of the development environment. That is, as long as the source HTML is given to WebChecker, how the HTML is generated does not matter.

Line 3 of Listing 2 uses *jsoup: Java HTML Parser* [9] library to parse the HTML file into *jsoup's Element* objects. While *jsoup* is very powerful, it is irrelevant in this context because a user does not need to learn a new library that is unrelated to a validation task. How the HTML file is parsed into objects should be done internally. WebChecker abstracts this by having the user only set the source of the HTML file as shown in Line 10 of Listing 3.

The *for-loop* and the *try-catch* blocks in Listing 2 are the least important to the user. WebChecker improves this by introducing a clear readable structure as explained in the WebChecker EVL Structure section above. Table I shows a side by side comparison of WebChecker with other current validators, purely based on our research and experiment.

Features	WebChecker	Other
Reusability	✓	
Configurable	✓	
Extensible	✓	
Platform Independence	✓	
Minimum Coding	✓	
Task Specific	✓	
Dynamic Page Analysis	✓	
Static Page Analysis	✓	✓

Table I: WebChecker Plugin Compared to Other Tools

VI. Related Work

Bootlint [10] is a Bootstrap specific linter for checking common HTML mistakes in web pages built in a vanilla way; that is, pages where default Bootstrap classes are used. Bootlint uses JavaScript to check for Bootstrap rule conformance, which is not configurable. The source code of Bootlint, which is in the *bootlint.js*[3] file of the Bootlint project, is much lengthier than the code in Listing 2. This validates our points on the current limitations of the current solution as explained above.

[3]https://github.com/twbs/bootlint/blob/master/src/bootlint.js

Policheck is Microsoft's internal tool that checks codes, comments, and contents including web pages. For example, it can check if a web page contains inappropriate content or its style matches best practices and guidelines [8].

ESLint [11] is a static JavaScript analysis linting utility written in JavaScript to enforce coding styles through pluggable rules. With ESLint, developers can analyze JavaScript code by using default rules or creating new rules without running the JavaScript code. ESLint requires a configuration file in JavaScript Object Notation (JSON), YAML (YAML Ain't Markup Language) or JavaScript. Most importantly, the configuration file includes the source of the rule file, which is a JavaScript file that has the implementation of the rule. While ESLint is powerful and popular, the JavaScript code for each rule is not reusable.

VII. Conclusion and Future Work

In this paper, we have proposed a new plugin for the Epsilon Validation Language with a concrete example. We identified the current solution developers use to check web pages for framework rule conformance, and compared our solution with the improvements it brings.

We believe that with web pages being ubiquitous, there will be more frameworks with implicit rules and less detailed documentations. If developers of these frameworks have a tool such as WebChecker, they can write exhaustive lists without providing much documentations since EVL constraints are easily readable and close to a natural language.

Further research on the topic includes extending WebChecker to support fixes when a constraint does not satisfy, evaluating other framework rules and capturing such implicit rules, providing WebChecker through popular text editors, validating web pages online, project-wise validation, and developing a domain specific language for capturing and enforcing such rules that is easy to integrate with a development workflow.

References

[1] "Bootstrap." "https://getbootstrap.com/".

[2] "Materialize." "https://materializecss.com/".

[3] "Foundation." "https://foundation.zurb.com/".

[4] "uikit." "https://getuikit.com/v2/index.html".

[5] "W3C ARIA in HTML." "https://www.w3.org/TR/html-aria/".

[6] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*, pp. 204–218. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[7] "Epsilon project." "https://www.eclipse.org/epsilon/".

[8] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 332–343, Sept 2016.

[9] "jsoup: Java html parser." "https://jsoup.org/".

[10] "Bootlint." "https://github.com/twbs/bootlint".

[11] "Eslint." "https://eslint.org/".