WebChecker: An Epsilon Validation Language Plugin

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

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

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) [1] plugin to validate static and dynamic HTML pages by writing configurable EVL constraints. WebChecker is used for enforcing implicit rules of HTML and CSS frameworks. To show the effectiveness of the plugin, we choose Bootstrap [2], the world's most popular HTML, CSS, and JavaScript framework. The plugin comes with an exhaustive list of EVL constraints for checking web pages built with Bootstrap. The bulk of the paper is devoted to WebChecker: what it is and how it is implemented.**

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. These rules are developed for better use of such frameworks. However, the caveat is to read their documentations carefully to capture such constraints and rules. This could be quite a tedious work. WebChecker is built to capture such rules so that consumers of these framework can easily check their web pages for the conformance of such rules.

The rest of the paper is structured as follows: Section II explains the problem and the current state of art solution and its limitation that motivated this project. In section III, we explain the WebChecker plugin, its syntax and features, and we show our solution for the problem. We compare the current solution and our solution of the problem and its improvements in Section IV. In Section V, we discuss related work. Lastly, Section VI 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 more than 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. 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 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 as follows: 1. Read 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<!— newCheck.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) [1]. 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.

```
File input = new File("files/bootstrap/newCheck.
    html");
try {
 Document doc = Jsoup.parse(input, "UTF-8");
 Elements elments = doc.getElementsByTag("div");
 for (Element element : elments) {
  if (element.hasClass("col-sm-4")) {
   if (!element.parent().hasClass("row")) {
    System.out.println("A div element with class
        col should have a parent element with
       class row");
   } else {
    if (!element.parent().parent().hasClass("
       container")) {
     System.out.println("A div element with
        class col should have a parent element
        with class row, which has a parent with
         class container.");
    }
   }
  }
 }
} catch (IOException e) {
 e.printStackTrace();
}
```

Listing 2: Enforcing Bootstrap Grid Rule

III. WebChecker Plugin

Epsilon Validation Language (EVL) [1] is one of the languages of Epsilon [3], 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 to access the underlying EVL features. Therefore, EVL users can use the features of WebChecker without setting up a new infrastructure 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 *separation of concern* principle with writing minimum code. In particular, there is an EVL source file and an HTML source, which could be an HTML file or its URI. The EVL file that contains the constraints such as Listing 4 is run against the HTML source. Listing 3 shows this connection. 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  //newCheck.java
2  //Step 1: Get the source to be validated
3  String html = "files/bootstrap/newCheck.html";
4
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 standard EVL structure.

- *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 and is used to enforce a rule. Each *constraint* have three sections *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 utilize this feature.
- *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*.

```
1 //newCheck.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

- *check*: this block *check*s 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 shown in Line 15 of Listing 4.

C. WebChecker Features

IV. Improvements over Current Solution

This section compares and evaluate WebChecker to the current solution.

V. Related Work

VI. Conclusion and Future Work

References

[1] 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.

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

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