Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving

Hesam Samimi*, Max Schäfer†, Shay Artzi†, Todd Millstein*, Frank Tip†, and Laurie Hendren‡
*Computer Science Department, University of California, Los Angeles, USA
{hesam, todd}@cs.ucla.edu
†IBM T.J. Watson Research Center, Hawthorne, NY, USA
{mschaefer, artzi, ftip}@us.ibm.com
‡School of Computer Science, McGill University, Montreal, Canada
hendren@cs.mcgill.ca

***Abstract*—PHP web applications routinely generate invalid HTML. Modern browsers silently correct HTML errors, but sometimes malformed pages render inconsistently, cause browser crashes, or expose security vulnerabilities. Fixing errors in generated pages is usually straightforward, but repairing the generating PHP program can be much harder. We observe that malformed HTML is often produced by incorrect *constant prints*, i.e., statements that print string literals, and present two tools for automatically repairing such HTML generation errors.** *PHPQuickFix* **repairs simple bugs by statically analyzing individual prints.** *PHPRepair* **handles more general repairs using a dynamic approach. Based on a test suite, the property that all tests should produce their expected output is encoded as a string constraint over variables representing constant prints. Solving this constraint describes how constant prints must be modified to make all tests pass. Both tools were implemented as an Eclipse plugin and evaluated on PHP programs containing hundreds of HTML generation errors, most of which our tools were able to repair automatically.**

***Keywords*-PHP; automated repair; string constraints**

I. Introduction

PHP is the most widely used server-side programming language for implementing web applications, with a recent survey finding that it is employed by about 77% of all websites [1]. Typically, a PHP application generates HTML pages based on user input and information retrieved from a database. These HTML pages often contain JavaScript code to enable interactive usage and links or forms referring to additional PHP scripts to be executed.

One particularly common issue plaguing many PHP applications is generation of invalid HTML. Modern browsers are quite tolerant of HTML errors and employ heuristics to silently correct them, although pages may render more slowly because of these error-correcting heuristics [2]. In some cases, however, erroneous HTML will be displayed differently depending on the browser, so the pages generated by a PHP program may look fine to the developer, while they would be unacceptable to a user with a different browser. In extreme cases, invalid HTML may even cause browsers to become unresponsive or expose security vulnerabilities [2]. Finally, erroneous HTML can be an obstacle to screen readers and other assistive technology. The World Wide Web Consortium maintains a collection of anecdotes from Web professionals about problems with malformed HTML[1].

This paper presents an approach to help programmers find and fix HTML generation errors in PHP programs. The approach is based on the observation that malformed HTML is most often generated due to errors in statements that print string literals. This is not surprising because such *constant prints* are the way in which a PHP program typically generates the tag structure of an HTML page. While, for example, the data in an HTML table might be generated via a dynamic database lookup in the PHP program, the table's HTML tags would be produced by printing the appropriate string literals (e.g., `"<tr>"` and `"<td>"`) in the right places.

We describe two tools for fixing HTML generation errors, which we have implemented in a plugin for the Eclipse PHP Development Tools.[2] The first tool, *PHPQuickFix*, targets a common special case, whereby a single constant print statement cannot possibly result in legal HTML. Examples include uses of HTML special characters such as '&' that should be escaped, or mismatched start and end tags as in `"<b><i>Yes!</b>"`. These problems are highlighted in the IDE and a quick-fix is suggested which, if accepted by the programmer, automatically repairs the affected code.

Our second tool, *PHPRepair*, targets the more general problem of incorrect constant print statements in PHP programs, including bugs caused by the interactions among multiple such statements and bugs that require adding, changing, or removing such statements. Given a test suite for a PHP program along with the expected HTML output for each test, we encode the condition that actual and expected output agree for each test case as a string constraint over variables corresponding to constant prints in the program. A string constraint solver automatically provides a solution

*Supported in part by National Science Foundation award CCF-0545850.

[1] See http://www.w3.org/QA/2009/01/valid_sites_work_better.html.

[2] See http://www.eclipse.org/pdt/.

to our constraint, which *PHPRepair* employs to modify the program appropriately. The result is a repaired program that passes all tests in the given suite.

PHPRepair only considers insertion, modification, and deletion of constant prints in a program. Despite the limited form of such repairs, they are still quite expressive, since the constant prints can be arbitrary; for example, a constant print may be inserted at any location in the program and there is no bound on the length of the string that it prints. Our focus on constant prints allows *PHPRepair* to perform an exhaustive search over the space of possible repairs, ensuring both *completeness* and *minimality*; prior work on automated program repair typically lacks these properties [3], [4].

In principle, a purely static analysis could give stronger guarantees than *PHPRepair*'s test-based approach. However, in practice the dynamic nature of PHP would make such an approach difficult to scale to real programs in a manner that only detects actual errors and fixes those errors without introducing new HTML generation bugs. In contrast, a testing approach is practical and effective for two main reasons. First, prior work has shown how to automatically generate high-coverage tests for PHP programs [2]. Second, while fixing a PHP program to correct HTML errors on all possible execution paths is quite challenging, fixing an individual broken HTML page is usually relatively straightforward, often requiring nothing more than, e.g., inserting a missing end tag. Indeed, such fixes are automatically suggested by tools such as HTML Tidy,[3] or they can be obtained by querying the DOM representation of the page inside a browser, which reflects the automatic corrections performed by the browser's HTML repair logic. Therefore, both a high-quality test suite and the associated test oracles required by our approach can be generated in a fully automatic manner.

We have evaluated our tools on several real-world PHP programs, showing that many HTML generation bugs can be fixed by our approach. *PHPQuickFix* identified several thousand shallow bugs that could be repaired by the suggested quick-fixes. Of the remaining bugs in each program, on average 86% were fixed by *PHPRepair*, which justifies our focus on constant print statements. A repair is found within seven seconds on average, so our tools are suitable for interactive use.

The remainder of the paper is organized as follows. In Section II we provide some background and introduce our two tools in the context of a motivating example. Section III precisely defines our notion of a repair and describes how the test-based tool *PHPRepair* finds repairs. Section IV provides implementation details for both tools. Section V evaluates our tools on a set of PHP programs, Section VI discusses related work and Section VII concludes.

```
 1 <html>
 2 <head>
 3   <title>List capitals</title>
 4   <style type="text/css">
 5     .highlight { background-color: Aquamarine; }
 6   </style>
 7 </head>
 8 <body>
 9   <?php
10     $highlight = isset($_GET["hl"]);
11     $con = mysql_connect("localhost", "test", "test");
12     mysql_select_db("countries", $con);
13     $data = mysql_query("SELECT * FROM countries");
14     if(!mysql_num_rows($data))
15       echo "<h1>No data!</h1>\n";
16     else {
17   ?>
18   <table border="2">
19   <tr><th><h3>Country</th><th><h3>Capital</h3></tr>
20   <?php
21       while($row = mysql_fetch_array($data)) {
22         echo "<tr><td>";
23         if($highlight)
24           echo "<div class='highlight'><b>";
25         echo $row['country'];
26         if($highlight) echo "</div></tr>";
27         else echo "</td>";
28         echo "<td>" . $row['capital'] . "</td>";
29         echo "</tr>\n";
30       }
31     }
32   ?>
33 </body>
34 </html>
```

Figure 1. A simple PHP script.

II. Background and Overview

A. An Example PHP Program

Figure 1 shows a small PHP script designed to illustrate our approach. The program queries a database for a list of countries and their capitals and renders this data as an HTML table, optionally highlighting country names by printing them in bold face on a light blue background.

A peculiar feature of PHP is that programs can contain fragments of inline HTML code that are printed verbatim when the program is executed. In the program of Fig. 1, there are several such fragments; the first one (lines 1–8) prints the page header including a CSS stylesheet, while the last one (lines 33–34) prints the page footer.

Snippets of PHP code appear inside `<?php` ... `?>` directives. The first snippet (lines 9–17) performs initialization and error checking: it uses the built-in function `isset` to determine whether the script was passed an HTTP GET parameter `hl`, setting flag `$highlight` accordingly; it then connects to a MySQL database containing the information to be displayed and sends a query to the database (lines 11–13).

If the query fails or returns no results, the body of the generated HTML page consists of the error message printed on line 15. Otherwise, another inline HTML fragment is used to emit the start tag of the table to be displayed (line 18)

[3] See http://tidy.sf.net.

Table I
Test cases for the script in Fig. 1.

ID	Database	Parameters	Output
t_1	$\emptyset$	$\emptyset$	see Fig. 2
t_2	countries = {(Canada, Ottawa), (Netherlands, Amsterdam), (USA, Washington)}	$\emptyset$	see Fig. 3
t_3	same as for t_2	{hl ↦ "1"}	see text

```
35 <html>
36 <head>
37   <title>List capitals</title>
38   <style type="text/css">
39     .highlight { background-color: Aquamarine; }
40   </style>
41 </head>
42 <body>
43 <h1>No data!</h1>
44 </body>
45 </html>
```

Figure 2. Valid HTML generated by the script in Fig. 1 on test case t_1.

and its first row containing the column headers.[4]

To build the table, the script iterates over the results of the query using a `while` loop (lines 21–30). For every query result, it prints a new row of the table, with two `td` elements containing the name of the country and its capital, respectively. If the script was passed the `hl` parameter, the country name is additionally wrapped in a `b` element to typeset it in bold font, and a `div` element with class `highlight`, which the CSS style sheet on line 5 styles using an aquamarine blue background.

B. HTML Generation Bugs

This example program contains several bugs similar to issues encountered in real-world PHP applications, which cause it to generate invalid HTML in certain situations. We will consider three test cases as described in Table I: t_1 runs the script on an empty database without setting parameter `hl`; t_2 uses a non-empty database containing information about the capitals of Canada, the Netherlands and the USA, but again does not set any parameters; and t_3 runs it on the same database as t_2 with parameter `hl` set to `"1"`.

In test case t_1, the program produces the HTML page in Fig. 2, which is syntactically correct.

In test case t_2, it produces the page in Fig. 3, which is not valid HTML: the first `h3` element on line 55 is missing an end tag, as is the `table` element on line 54.[5]

```
46 <html>
47 <head>
48   <title>List capitals</title>
49   <style type="text/css">
50     .highlight { background-color: Aquamarine; }
51   </style>
52 </head>
53 <body>
54   <table border="2">
55   <tr><th><h3>Country</th><th><h3>Capital</h3></tr>
56   <tr><td>Canada</td><td>Ottawa</td></tr>
57   <tr><td>Netherlands</td><td>Amsterdam</td></tr>
58   <tr><td>USA</td><td>Washington</td></tr>
59 </body>
60 </html>
```

Figure 3. Invalid HTML generated by the script in Fig. 1 on test case t_2.

These two problems are silently corrected by modern browsers, allowing the page to display as intended. For instance, inspection of the DOM produced when this page is displayed in Google Chrome 13.0 shows that it inserts the missing end tags as expected: a `</h3>` tag before the `</th>` tag on line 55, and `</table>` before the `</body>` tag on line 59.

Translating these fixes for the generated HTML page into fixes for the generating PHP program can be much more difficult, however. It is clear that the first `h3` element in the inline HTML on line 19 must be closed by adding `</h3>` on that same line between `Country` and `</th>`. However, there are many possible options for inserting the missing statement `echo "</table>"` to close the start tag on line 18, and it is easy to do so improperly. If it is inserted as part of that same block of inline HTML, then the table will be closed before all of its rows have been output. If it is inserted inside the body of the while loop, the result on test t_2 will be to emit three `</table>` tags. Inserting it after line 31 would repair test t_2 but would break test t_1. Inserting it after line 30 leads to valid HTML being produced in both cases.

Finally, consider test t_3, where the script is run on the same database as in t_2, but now with the parameter `hl` set to `"1"`. This produces essentially the same page as in Fig. 3, but the table rows are now of the following form:

```
<tr><td><div class='highlight'><b>Canada</div></tr>
    <td>Ottawa</td></tr>
```

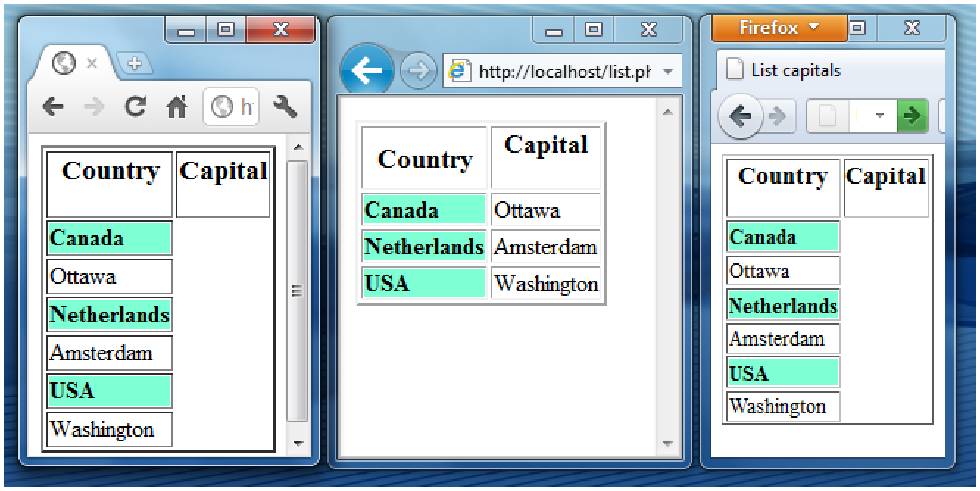
This is not valid HTML: the `b` element is missing an end tag, and a `</tr>` tag occurs where a `</td>` tag is expected.

Different browsers display this invalid HTML page in different ways, as shown in Fig. 4. While they all insert the missing `</b>` tag before the `</div>` tag, the unexpected `</tr>` tag is not handled uniformly. Internet Explorer decides to treat it as a `</td>` tag, which is arguably the best fix from a user's perspective; Google Chrome and Firefox, on the other hand, silently insert a `</td>` tag before the `</tr>` and another `<tr>` tag after it, thus splitting one row into two and upsetting the table layout considerably. These kinds of inconsistencies, which can occur across different browsers as well as different versions of the same browser, are very easy for developers to miss during testing.

As before, propagating the desired HTML fixes (in this case, those performed by Internet Explorer) back to the gen-

[4] Note that this HTML fragment is printed as part of the `else` branch of the `if` statement on line 14, which is only closed on line 31 in another PHP snippet: PHP code and HTML fragments can be freely mixed without regard to syntactic nesting, and this is frequently done in real-world programs.

[5] Perhaps surprisingly, the missing end tag of the second `th` element on line 55 is not a problem: `th` is self-closing, hence its end tag is optional.



Country	Capital
Canada	
Ottawa	
Netherlands	
Amsterdam	
USA	
Washington	

Country	Capital
Canada	Ottawa
Netherlands	Amsterdam
USA	Washington

Country	Capital
Canada	
Ottawa	
Netherlands	
Amsterdam	
USA	
Washington	

Figure 4. Different renderings of our invalid HTML page in Google Chrome 13.0 (left), Internet Explorer 9.0 (middle) and Firefox 6.0 (right).

```
63 <html>
64   <head>
65     <style type="text/css">
66       #test-div { margin:0 10px 10px; }
67       #test-form { width:100%; }
68     </style>
69   </head>
70   <body>
71     <table>
72       <tr><th>Test:</th></tr>
73       <tr><td><div id="test-div">
74             <form id="test-form" method="post">
75               <input type="text" name="test"/>
76           </div></td></tr>
77     </table>
78   </body>
79 </html>
```

Figure 5. An invalid HTML page that causes Internet Explorer to hang.

erating PHP program is non-trivial. Of the various options for places to insert the missing `</b>` tag, the right place is on line 26 in the "then" branch of the `if` statement. Emitting it before the `if` statement would be acceptable for this test case but would break t_2. Propagating the other fix requires modifying the `</tr>` tag on the same line to `</td>`.

Problems such as incorrect or missing end tags may seem trivial, but they by no means always are: the HTML page in Fig. 5, adapted from http://crashie8.com, causes even very recent versions of Internet Explorer to hang while displaying without problems in other browsers. The problem here is a missing end tag for the `form` element starting on line 74 which, in a somewhat subtle combination with a table and a CSS stylesheet, triggers a bug in the browser's HTML repair logic. In some cases, invalid HTML can also impact browser performance, or lead to security vulnerabilities [2].

C. Automated PHP Program Repair

While repairing a PHP program can in principle require arbitrary modifications to its statements and structure, we observe that repairing HTML generation bugs often requires only additions, modifications, and removals of statements that print string literals (e.g., inline HTML or an **echo** or **print** statement whose argument is a string literal), which we collectively dub *constant prints*. This is the case because, as illustrated in Fig. 1, constant prints are the mechanism by which the tag structure of an HTML page is generated. By focusing on this common class of repair actions, we have devised a two-pronged approach to automatically repairing PHP programs that is simple yet effective, and have implemented the approach as a plugin for the Eclipse PHP Development Tools.

First, the fragile nature of PHP results in many shallow HTML generation bugs, whereby a single constant print is erroneous in the sense that it cannot possibly result in legal HTML, no matter what context it is executed under. For instance, line 19 of the program in Fig. 1 can be seen to be erroneous in isolation, since it contains improperly nested HTML start and end tags. We have developed *PHPQuickFix*, a static checker that catches these kinds of local HTML generation bugs. *PHPQuickFix* attempts to parse each string constant and inline HTML fragment in the program in isolation. While parsing a string, *PHPQuickFix* maintains a stack of currently open elements; whenever it detects an end tag that does not match the most recently opened tag, it generates a quick-fix proposal to insert the expected end tag at the current location. The tool also identifies and generates a quick-fix for two simple but common errors, namely ampersand characters that are not part of a character reference (and hence should be escaped as `"&amp;"`), and non-alphanumeric attribute values that are not quoted.

Since *PHPQuickFix* considers each constant print in isolation, it cannot detect or repair HTML errors involving multiple program points, such as the missing `</table>` tag in our running example. Generalizing *PHPQuickFix* to perform static analysis over an entire program would be quite difficult due to the many highly dynamic features of the language and the need to precisely model the effect of varying databases and parameter settings on control and data flow.

Instead, we propose a test-based approach to repairing complex HTML generation bugs. Our approach assumes that a test suite for the program is available. Each test in this suite is described by: (i) the input data on which to run the program (such as HTTP parameters and databases), and (ii) the expected output the program is supposed to produce. Such a test suite can be produced without user interaction by employing a high-coverage test generation tool for PHP such as Apollo [2] and an HTML repair tool such as HTML Tidy, or it can be created manually. Our second tool, *PHPRepair*, automatically adds, modifies, and removes constant prints in the given PHP program in order to produce a program that passes all tests in the given suite.

PHPRepair is based on the idea that we can characterize a given test's execution by the sequence of strings output by individual print statements that are executed in the program, say $s_1, \ldots, s_n$. If $s_1 \cdot \cdots \cdot s_n = e$, where "." represents string concatenation and e is the expected output, then the test case passes, otherwise it fails. Replacing each s_i that results from a constant print with a constraint variable v_i in the above equation encodes a string constraint whose solution tells us how to repair the program to satisfy the test case. A solution to all the constraints generated from a test suite leads to a repair that makes the whole suite pass.

In the next section, we develop this basic idea in more detail on the basis of the examples in this section.

III. Input-Output Based Repair

A. Test Cases and Repairs

A program p is a collection of PHP scripts. A *test case* $t = (\rho, \sigma)$ consists of a configuration ρ to run the subject program under and an expected output σ. For the purposes of this discussion, the precise structure of ρ is irrelevant; it could, for instance, specify an initial database configuration, a sequence of scripts to execute, and the values of HTTP parameters to pass to the scripts. The *actual output* p produces on t is the HTML page generated by the last script invoked when running p under ρ.[6] The program is said to *pass* test case t if the actual output of p on t equals the expected output σ.

We refer to an inline HTML fragment or a `print` or `echo` statement as a *print*. A print whose argument is a string literal is called a *constant print*, or cprint for short. Any other print is called a *variable print* or vprint.

Programs p and p' are called *repair convertible* if one can be obtained from the other by repeatedly performing any of the following repair actions: (i) adding a new cprint, (ii) removing a cprint, or (iii) modifying an existing cprint (by changing the string constant that it prints). A *repair problem* consists of a program p and a set T of tests. A solution of the repair problem is a program p' such that p and p' are repair convertible and p' passes all tests in T.

```
<html>
<head>
  <title>List capitals</title>
  <style type="text/css">
    .highlight { background-color: Aquamarine; }
  </style>
</head>
<body>
  <table border="2">
<tr><th><h3>Country</h3></th><th><h3>Capital</h3></tr>
  <tr><td>Canada</td><td>Ottawa</td></tr>
  <tr><td>Netherlands</td><td>Amsterdam</td></tr>
  <tr><td>USA</td><td>Washington</td></tr>
  </table>
</body>
</html>
```

Figure 6. Expected output for test t_2 (non-empty database, no parameters).

```
<html>
<head>
  <title>List capitals</title>
  <style type="text/css">
    .highlight { background-color: Aquamarine; }
  </style>
</head>
<body>
  <table border="2">
<tr><th><h3>Country</h3></th><th><h3>Capital</h3></tr>
  <tr><td><div class='highlight'><b>Canada</b></div>
    </td><td>Ottawa</td></tr>
<tr><td><div class='highlight'><b>Netherlands</b></div>
    </td><td>Amsterdam</td></tr>
  <tr><td><div class='highlight'><b>USA</b></div>
    </td><td>Washington</td></tr>
  </table>
</body>
</html>
```

Figure 7. Expected output for test case t_3 (non-empty database, parameter `hl = "1"`).

Note that we only consider repair actions involving cprints. In particular, we do not consider adding, deleting or modifying vprints, or changing the program's control structure. Our evaluation in Sec. V suggests that most real-world HTML generation bugs can be repaired in this way.

As an example of a repair problem, consider the program of Fig. 1 and the test suite $T = \{t_1, t_2, t_3\}$ consisting of the test cases described in Table I, which each only invoke a single script (the one shown in Fig. 1). The expected output for t_1 is the same as the actual output, shown in Fig. 2; the expected outputs for t_2 and t_3 are given in Fig. 6 and Fig. 7.

Fig. 8 shows the repairs to be performed to solve this repair problem, where changes are highlighted and unchanged portions of the program are omitted: two existing cprints are modified, and one new cprint is added.

[6] While earlier scripts do not directly contribute to the actual output, they may alter the database or session state, and hence indirectly influence it.

```
...
19  <tr><th><h3>Country </h3> </th><th><h3>Capital</h3></tr>
...
26          echo " </b> </div></t d >";
...
31      echo "</table>"; }
...
```

Figure 8. Repair for the PHP script in Fig. 1.

B. Properties

We have designed an approach that sets up a constraint system to capture the semantics of the repair problem as defined above, with solutions representing repairs. Before discussing it in detail, let us consider what properties we desire from such an approach.

1) *Soundness*: If the constraint system has a solution, it should represent a valid repair, i.e., the repaired program should pass every test in the suite.
2) *Completeness*: If a valid repair exists, the constraint system should have a solution.
3) *Minimality*: For usability, we would like to find a repair that is minimal in the sense that it modifies the original program as little as possible.

Our approach makes two assumptions about the given program and its test suite. Firstly, the program may not inspect or modify its own source code; this is needed since we rely on source-level instrumentation to dynamically collect information about program executions. Secondly, all tests must be deterministic, i.e., the program must execute in the same way (and in particular produce the same output) every time it runs a given test. Since individual PHP scripts are not usually interactive this is not a severe restriction.

C. Finding a Sound Repair

Let a program p and a test suite T be given. If we assign a unique label to every print in p, we can characterize an execution of p on a test $t \in T$ by its *print trace*, which is the sequence of prints encountered during the execution together with the string values they printed.

Figure 9 shows a possible labeling of the script from Fig. 1, where we have labeled the cprints as c_1 to c_{14}, and the vprints as v_1 and v_2. Multi-line HTML fragments are counted as a single cprint and only get a single label. Additional empty cprints have been inserted on lines 131, 133, and 141; these are necessary for the completeness of our approach and are explained in more detail below.

Using this labeling, the print trace of running the program on test t_1 is

$$[(c_1, \texttt{"<html>}\ldots\texttt{"}), (c_2, \texttt{"<h1>}\ldots\texttt{"}), (c_{14}, \texttt{"</body>}\ldots\texttt{"})]$$

reflecting the fact that the program executed the cprints on lines 115 - 117, 121, and 144 - 145 (in this order), but no vprints.

```
115 <html>
116 ...
117 <body>^c1
118   <?php
119     ...
120     if(!mysql_num_rows($data))
121       echo "<h1>No data!</h1>\n"^c2;
122     else {
123   ?>
124   <table border="2">
125   <tr><th><h3>Country</th><th><h3>Capital</h3></tr>^c3
126   <?php
127       while($row = mysql_fetch_array($data)) {
128         echo "<tr><td>"^c4;
129         if($highlight)
130           echo "<div class='highlight'><b>"^c5;
131         echo ""^c6;
132         echo $row['country']^v1;
133         echo ""^c7;
134         if($highlight) echo "</div></tr>"^c8;
135         else echo "</td>"^c9;
136         echo "<td>"^c10;
137         echo $row['capital']^v2;
138         echo "</td>"^c11;
139         echo "</tr>\n"^c12;
140       }
141       echo ""^c13;
142     }
143   ?>
144 </body>
145 </html>^c14
```

Figure 9. Labeled version of the script from Fig. 1.

Since a cprint will print the same string every time it is executed, we can abbreviate print traces by omitting the output of cprints. Using this convention, the print trace of test t_2 is as follows (and is similar for t_3).

$$\begin{aligned}&[c_1, c_3,\\ &c_4, c_6, (v_1, \texttt{"Canada"}), c_7, c_9, c_{10}, (v_2, \texttt{"Ottawa"}), c_{11}, c_{12},\\ &c_4, c_6, (v_1, \texttt{"Ne...\!"}), c_7, c_9, c_{10}, (v_2, \texttt{"Amsterdam"}), c_{11}, c_{12},\\ &c_4, c_6, (v_1, \texttt{"USA"}), c_7, c_9, c_{10}, (v_2, \texttt{"Washington"}), c_{11}, c_{12},\\ &c_{13}, c_{14}]\end{aligned}$$

Clearly, the program passes a test case if the concatenation of all the output strings in the associated print trace equals the expected output. If we interpret the labels of cprints as constraint variables, we can express the condition that actual output and expected output on a test case must agree as a string constraint: the left hand side of the constraint is the concatenation of all labels in the print trace, while the right hand side is simply the expected output. We will call this constraint the *repair constraint*.

The repair constraint for test case t_1, for instance, is

$$c_1 \,.\, c_2 \,.\, c_{14} = \sigma_1$$

where σ_1 is the HTML document of Fig. 2. One solution to the constraint has c_1, c_2, and c_{14} take on their original values, i.e., the string literals printed by the corresponding cprints in the original program.

Since our approach to program repair only attempts to modify constant prints, we do not represent vprints as constraint variables. Indeed, using a constraint variable for a

Table II
A SOLUTION FOR THE REPAIR CONSTRAINTS ENCODING t_1, t_2 AND t_3.

Var	Old value	Repair value
c_3	"...Country</th>..."	"...Country</h3></th>..."
c_8	"</div></tr>"	"</b></div></td>"
c_{13}	""	"</table>"

vprint would in general lead to unsolvable constraints, since a single vprint may produce a different output each time it is executed (e.g., v_1 in Fig. 9). Instead, we represent each occurrence of a vprint by the (constant) output it produced in the execution in question. For test case t_2, we then obtain the repair constraint

$$\begin{aligned}
& c_1 . c_3 . \\
& c_4 . c_6 . \texttt{"Canada"} . c_7 . c_9 . c_{10} . \texttt{"Ottawa"} . c_{11} . c_{12} . \\
& c_4 . c_6 . \texttt{"Netherlands"} . c_7 . c_9 . c_{10} . \texttt{"Amsterdam"} . c_{11} . c_{12} . \\
& c_4 . c_6 . \texttt{"USA"} . c_7 . c_9 . c_{10} . \texttt{"Washington"} . c_{11} . c_{12} . \\
& c_{13} . c_{14} = \sigma_2
\end{aligned}$$

where σ_2 is the HTML page in Fig. 6. This constraint, as well as the one above for test t_1, is solved by setting $c_3 := \texttt{"...Country</h3></th>..."}$, $c_{13} := \texttt{"</table>"}$, and all other variables to their original values.

A satisfying assignment for a set of repair constraints directly corresponds to a repair in which every cprint is modified to print the string assigned to its constraint variable. The repaired program will then by construction pass the test cases encoded by the constraints. Therefore, a solution to the repair constraints for a given test suite corresponds to a repair that makes the program pass every test in the suite. Table II shows a solution for the repair constraints encoding tests t_1, t_2 and t_3 (omitting unchanged variables), corresponding to the repair in Fig. 8.

D. *Ensuring Completeness and Minimality*

While the approach outlined so far is sound if the underlying constraint solver is, it can only find repairs involving modifications of existing cprints (including setting the string of a cprint to the empty string, which is tantamount to deletion). There is no support for adding new cprints, hence the approach is not yet complete.

Note that it is never necessary to add a new cprint c' right before or after an existing cprint c: instead of adding c' we can just as well modify c. For the same reason, it is unnecessary to add c' if it is in the same block of straight-line code as c and there are no vprints in between. Thus, our approach is complete if the program to be repaired has a cprint at the beginning of every code block, after every vprint, and after every nested code block.

We can easily bring any program into this form by padding it with trivial cprints of the form `echo ""`, and *PHPRepair* performs this simple modification. For instance, in the program of Fig. 9, cprints are inserted on lines 131 and 141 after nested blocks, and on line 133 after a vprint. On the other hand, there is no need to insert a cprint after line 127 at the beginning of the loop body, as there already is a cprint on the next line.[7]

To achieve minimality, we use a cost metric to characterize the number of changes required by a repair. Let M be a solution for the set of repair constraints under consideration. Then we can define $\mathrm{cost}(M)$ as the number of variables to which M assigns a different value than its original value, meaning that M is considered more expensive the more cprints it modifies. In order to find a minimal repair for the program, we then simply look for a solution with the minimum cost. While this cost metric assigns the same cost to every modification, it is easy to substitute a different metric that, for instance, penalizes adding a cprint more heavily than modifying an existing one, or takes the amount of change to each print statement into consideration.

IV. IMPLEMENTATION

In this section, we discuss implementation details of our two repair tools. The tools are part of a plugin that we have built for the Eclipse PHP Development Tools (PDT).

A. *PHPQuickFix*

PHPQuickFix integrates into the PDT as a build participant and is run every time a source file is processed. Warnings are displayed as annotations in the IDE and come with a proposed quick-fix that, if the user accepts it, modifies the program directly in the editor to repair the problem.

As described in Section II-C, *PHPQuickFix* checks each string literal in the program in isolation for common HTML generation errors. This is unsound in general, since some strings do not represent HTML or undergo further processing before being printed. Hence *PHPQuickFix* sometimes emits spurious warnings, and the programmer has to exercise caution in applying the suggested fixes. Since it only considers one string literal at a time, *PHPQuickFix* is of course also not complete. Nevertheless, our experiments in Section V indicate that *PHPQuickFix* is a valuable tool for quickly detecting and eliminating common HTML generation bugs.

B. *PHPRepair*

PHPRepair can be invoked via a menu item added to the PDT. The option requires the developer to specify an XML file which contains an encoding of the test suite.

PHPRepair first uses source-level instrumentation to generate the repair constraints. From the original program p it creates an instrumented program p_I that is identical to p except that trivial cprints are inserted as described in Sec. III-D and all prints are replaced by calls to a logging function, which performs the normal print and logs both the label of the print and the output it produces. Running p_I

[7] An inserted cprint will not actually appear in the repaired program unless the solution to the repair constraints requires it.

Table III
DIFF REGIONS FOR THE EXAMPLE TEST SUITE.

Tests	Actual Output	Expected Output	Diff variables
t_2, t_3	...Country </t...	...Country </h3> </t...	c_3
t_2, t_3	...</tr> </b	...</tr> </table> </b...	c_{13}
t_3	...ada </div></tr> <td...	...ada </b></div></td> <td...	c_7, c_8
t_3	...nds </div></tr> <td...	...nds </b></div></td> <td...	c_7, c_8
t_3	...USA </div></tr> <td...	...USA </b></div></td> <td...	c_7, c_8

on a test case produces a log containing the associated print trace, from which the repair constraint is constructed.

PHPRepair then solves the set of repair constraints by encoding them in the input language of Kodkod [5], an efficient SAT-based constraint solver. We considered using an off-the-shelf string constraint solver such as Hampi [6] or Kaluza [7] instead, but neither solver supports cost optimization, which we need in order to find a minimal repair. In contrast, Kodkod can be easily used with any underlying SAT solver, including a cost-optimizing one.

Another advantage of Kodkod is that it provides a simple way to bound the allowed solutions to each variable, which we use to drastically reduce the search space. Such bounds are easy to obtain due to the simple form of our repair constraints, where the right-hand sides are constant. For example, the string value of a variable v appearing in a constraint C must be entirely composed of characters appearing on the right-hand side of C, and its maximum length is bounded by the length of the right-hand side.

We further optimize the constraints passed to the solver by employing a simple *localization* heuristic, based on the observation that the differences between the actual and expected outputs for failing test cases are generally small. We first compute *diff regions* for each test case, i.e., substrings of the actual output that do not agree with the expected output. Using the logged print traces, we can then identify all cprints that produce output in a diff region. We call the variables corresponding to these cprints *diff variables*. In the example of Sec. II, the actual and expected output on tests t_2 and t_3 yield five diff regions shown in Table III. The last column lists the diff variables for every region; overall, the diff variables for this test suite are c_3, c_7, c_8, c_{13}.

Given this information, our heuristic forces all non-diff variables to retain their original values, since they do not contribute to any of the diff regions and hence likely already have their correct values. We do this by canceling out each non-diff variable from the left-hand side of each repair constraint, along with its corresponding expected output on the right-hand side (which by definition matches the variable's actual output). The result is a set of localized constraints in place of each original repair constraint. From the repair constraint for test t_2 we get three localized constraints

$$\begin{aligned} c_3 &= \texttt{"...Country</h3></th>..."} \\ c_7 &= \texttt{""} \\ c_{13} &= \texttt{"</table>"} \end{aligned}$$

whereas t_3 contributes only one new constraint:

$$c_7 \,.\, c_8 = \texttt{"</b></div></td>"}$$

These four constraints can be solved by Kodkod, leading to the solution shown in Table II.

Localization is sound and critical in practice for reducing repair time, but it can sometimes lose solutions since it does not allow a cprint outside of a diff region to be modified. For example, consider a repair that requires hoisting a cprint from within an `if` block to occur just before the conditional. If the block is only executed on passing tests, our heuristic will not allow that cprint to be modified, causing the localized constraints to become unsatisfiable. We regain completeness through a simple back-off procedure: if the localized constraints are unsatisfiable, we expand each diff region by a fixed amount and try again. In the limit, each test output becomes a single diff region, causing the original repair constraints to be solved.

Finally, we observe that constraints that do not have any variables in common can be solved independently. We can hence improve constraint solving time by partitioning the repair constraints according to their variables and solving each partition separately. This optimization is particularly effective after localization, which tends to produce many constraints that each refer to a small number of variables.

V. EVALUATION

We present an evaluation of our repair techniques on a set of PHP applications, focusing on three evaluation criteria:

EC1 How many HTML generation errors can *PHPQuickFix* fix? How many spurious warnings does it produce?

EC2 How successful is *PHPRepair* in repairing the remaining HTML generation errors?

EC3 When *PHPRepair* fails, how often is this due to the restriction to modify only cprints and how often due to limitations of the constraint solver?

Table IV
SUBJECT PROGRAMS.

program	version	# files	LOC	# tests	coverage
faqforge	1.3.2	19	734	536	89.2%
webchess	0.9.0	24	2,226	979	40.6%
schoolmate	1.5.4	63	4,263	676	65.5%
hgb	4.0	20	541	1359	97.2%
timeclock	1.0.3	62	13,879	958	26.8%
dnscript	N/A	60	1,156	1,167	75.9%

A. Experimental Setup and Methodology

Table IV describes our subject programs and their test suites. The LOC column lists the number of lines containing an executable PHP statement, and the last two columns give the size of the test suite and its line coverage. The tests were generated automatically using Apollo [2] on a time budget of 20 minutes. Coverage for *timeclock* is low since it makes heavy use of client-side JavaScript, which is not very well supported by Apollo. Note that each test typically triggers multiple HTML generation errors, and a single error may be triggered by multiple tests, so the number of failing tests tends to be correlated only loosely with the number of bugs.

PHPRepair additionally requires the expected output for each test. We used the W3C Markup Validation Service[8] to identify validity violations and HTML Tidy to automatically fix simple HTML errors. More complex errors that exceed the capabilities of HTML Tidy were fixed by hand. To address criterion EC3, we also manually constructed "golden" versions of the subject programs that produce the expected output on all tests. This required significant effort on the order of several days of work for the larger benchmarks.

On each benchmark, we first used *PHPQuickFix* to fix all the simple HTML generation errors and then applied *PHPRepair* to the modified program and its test suite to repair more complex errors. We believe this approach reflects the way in which our tools would be used by programmers.

To simulate a developer interacting with *PHPRepair* to repair all failing tests in a test suite $t_1, \ldots, t_n$, we used the following iterative process. Let t_f be the first failing test case. We first run *PHPRepair* on tests $t_1, \ldots, t_f$, with a timeout of three minutes for the solver. Recall from the end of Sec. IV that we partition the repair constraints into independent sets. We automatically apply to t_f the repairs corresponding to each constraint set for which *PHPRepair* provides a solution. If all sets have solutions, then test t_f has been fully repaired so we move on to the next failing test case. Otherwise we manually apply as many fixes from the "golden" version as required to make t_f pass before moving on. We repeat these steps until all tests pass.

To measure the effectiveness of our approach we count the total number of *patches* (i.e., positions where a contiguous program fragment was inserted, modified, or removed) required to fully repair each program, and compute what percentage of patches were applied automatically by *PHPRepair*. This is a more objective metric than the number of fixed test cases, which depends heavily on the test suite. Using the number of validator error messages is also problematic, since a single error may lead to several messages.

Table V
NUMBER OF ERRORS FOUND AND REPAIRED BY *PHPQuickFix*.

name	# errors reported	# false positives
faqforge	92	0
webchess	4	0
schoolmate	44	0
hgb	189	139
timeclock	2733	5
dnscript	22	1

Table VI
NUMBER OF ERRORS FOUND AND REPAIRED BY *PHPRepair*.

name	# tool patches	# all manual patches	# non-cprint manual patches	% tool patches
faqforge	33	3	3	92%
webchess	4	0	0	100%
schoolmate	11	0	0	100%
hgb	88	40	6	69%
timeclock	315	64	55	83%
dnscript	74	25	6	75%

B. Results

EC1: Table V shows the number of HTML generation errors reported by *PHPQuickFix* on each of the subject programs in the middle column. As can be seen from these results, the simple kinds of errors detected by *PHPQuickFix* are quite common. As mentioned in Sec. IV, *PHPQuickFix* can incur false positives, but on most of our benchmarks this is very rare. The sole exception is *hgb*, which uses custom HTML templates with un-escaped ampersand characters as field separators. These characters are substituted away when producing actual HTML output, but the local analysis performed by *PHPQuickFix* cannot recognize this.

EC2: Table VI reports on our evaluation of *PHPRepair*, listing for every benchmark the number of patches automatically applied by *PHPRepair* and the number of patches applied manually; the fourth column shows how many of the latter involved fixing statements other than cprints and hence exceeded the capabilities of our tool. Across all benchmarks, *PHPRepair* on average performs 86% of all patches automatically. On these benchmarks, our iterative process went through a total of 125 iterations. On 42 iterations, *PHPRepair* timed out without finding a solution; the remaining iterations completed in an average of 7 seconds.[9]

The relatively low percentage of automated repairs on *hgb* is largely an artifact of our evaluation strategy: most manual patches could have been found automatically, but they occurred in the same test case (and constraint) as a more complicated repair and so had to be applied by hand.

[8] See http://validator.w3.org/.

[9] Measured on a 2.4GHz Core 2 Duo Macbook Pro with 2 GB of RAM.

EC3: The fourth column in Table VI shows that on average only about 6% of the necessary repairs were out of scope for our approach. Examples of such repairs are missing `include` statements and faulty vprints. The high number of non-cprint patches in *timeclock* is due to a single patch involving a vprint that is required in every script.

While most of the invalid HTML generated by our benchmarks would be silently corrected by a browser, we found three errors that resulted in visible layout problems, two of which were automatically fixed by *PHPRepair*.

C. Threats to Validity

The subject programs used in our evaluation may not be representative of other PHP programs. We did not specifically select the benchmarks to suit our approach; many of them have been used in our previous research [2]. Some PHP programs (such as *phpBB2* [2]) use custom templating mechanisms to generate their output, whereby a template of the page to generate is read from a file and subjected to some string processing to generate the actual output page. Our approach does not work well on such programs, which typically contain few cprints.

The bugs we detected and fixed may not be representative since the test suites we use do not cover all of a program's behavior. However, the test suites achieve high coverage and were generated using algorithms that are completely unrelated to the repair techniques studied in this paper.

Finally, there is often more than one way to fix a given HTML generation error, but in our evaluation we had to pick a single fix. When constructing the corrected HTML output and the golden versions of our subject programs, we have attempted to choose "sensible" repairs that disturb the original structure as little as possible.

VI. Related Work

Static analysis of strings in web applications has been used to validate HTML output from web applications [8], [9], to ensure that only XML documents meeting a given DTD are generated [10], and to detect security vulnerabilities [11], [12], [13]. Our *PHPQuickFix* tool also performs a static analysis, but only handles the special case of HTML errors within an individual string literal. Since *PHPQuickFix* is neither sound nor complete it cannot guarantee the absence of all errors; similarly, *PHPRepair* is only sound up to the given test suite. However, our tools can automatically repair HTML generation errors, rather than simply identifying them. Due to its dynamic approach, *PHPRepair* does not incur false positives as a static tool might.

Nguyen *et al.* [4] tackle the same problem of repairing HTML generation errors in PHP code, but in a very different way. They use a heuristic algorithm to map HTML output back to the program, while we use instrumentation to get a precise mapping. Like us they focus on constant prints, but their heuristic repair algorithm does not appear to ensure soundness, completeness or minimality. Finally, their evaluation only considers fixes found by HTML Tidy; we also consider more complicated manual fixes.

Weimer et al. [3] use genetic programming to repair C programs, whereby repairs are found by adapting statements from other locations in a program. Like ours, their approach requires a test suite, uses instrumentation to record execution paths, and guarantees correctness up to that suite. Our focus on constant prints allows us to perform exhaustive search for repairs, ensuring both completeness and minimality. Genetic programming approaches support more complex repairs but rely on heuristics and hence lack these important properties.

There has also been work on synthesizing programs that meet a given specification. Closest to our work are approaches that require the user to provide an initial program template with "holes" to be filled in [14], [15]. *PHPRepair* implicitly allows any cprint as a "hole" and uses tests to identify which ones to modify along with cost minimization to avoid unnecessary patches. Finally, Gulwani [16] described a tool to synthesize Excel spreadsheet macros. Like *PHPRepair*, that approach is based on input-output examples and synthesizes a program that generates strings. However, programs are synthesized in a specialized domain-specific language, while we repair arbitrary PHP programs.

Angelic debugging [17], like our approach, uses constraint solving over a test suite to identify erroneous expressions. While it can handle more general errors, angelic debugging is in general not able to suggest source-level repairs.

Several projects use constraint solving for automatic program transformations, often in the form of refactorings, as in type-related refactorings [18], refactoring for inferring generic types in Java [19], and refactorings that manipulate access modifiers [20].

VII. Conclusions and Future Work

We have presented a novel approach to automatically repair HTML generation errors in PHP programs, targeting a common class of repairs based on adding, modifying, and removing statements that print string literals. We have developed a simple static tool, *PHPQuickFix*, for repairing errors local to a single print statement, and a test-based tool, *PHPRepair*, for repairing more complex errors by solving a system of string constraints. Our experiments show that these tools are able to efficiently repair most HTML generation bugs in a variety of open-source benchmark programs.

There are several avenues for further research. We would like to experiment with different cost metrics incorporating knowledge of the program's structure (e.g., to encourage solutions where all fixes are localized in the same script). To improve performance, we may be able to leverage the highly structured form of our constraints to aggressively optimize our SAT-based encoding, rather than relying on Kodkod's built-in encoding. Finally, we would like to generalize our approach to handle more complex repairs.

References

[1] W^{3}Techs, “Usage Statistics and Market Share of PHP for Websites,” http://w3techs.com.

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst, “Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking,” *IEEE TSE*, vol. 36, no. 4, pp. 474–494, 2010.

[3] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, “Automatically Finding Patches Using Genetic Programming,” in *ICSE*, 2009, pp. 364–374.

[4] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Auto-Locating and Fix-Propagating for HTML Validation Errors to PHP Server-Side Code,” in *ASE*, 2011, pp. 13–22.

[5] E. Torlak, “A constraint solver for software engineering: Finding models and cores of large relational specifications,” Ph.D. dissertation, MIT, 2009.

[6] V. Ganesh, A. Kieżun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection,” in *CAV*, 2011, pp. 1–19.

[7] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A Symbolic Execution Framework for JavaScript,” in *IEEE Symp. on Security and Privacy*, 2010, pp. 513–528.

[8] Y. Minamide, “Static Approximation of Dynamically Generated Web Pages,” in *WWW*, 2005, pp. 432–441.

[9] A. Møller and M. Schwarz, “HTML Validation of Context-Free Languages,” in *FOSSACS*, 2011, pp. 426–440.

[10] Y. Minamide and A. Tozawa, “XML Validation for Context-Free Grammars,” in *APLAS*, 2006, pp. 357–373.

[11] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, “Static Checking of Dynamically Generated Queries in Database Applications,” *ACM TOSEM*, vol. 16, September 2007.

[12] G. Wassermann and Z. Su, “Static Detection of Cross-Site Scripting Vulnerabilities,” in *ICSE*, 2008, pp. 171–180.

[13] F. Yu, M. Alkhalaf, and T. Bultan, “Patching Vulnerabilities with Sanitization Synthesis,” in *ICSE*, 2011, pp. 251–260.

[14] A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat, “Combinatorial Sketching for Finite Programs,” in *ASPLOS*, 2006, pp. 404–415.

[15] A. Solar-Lezama, C. G. Jones, and R. Bodík, “Sketching Concurrent Data Structures,” in *PLDI*, 2008, pp. 136–148.

[16] S. Gulwani, “Automating String Processing in Spreadsheets Using Input-Output Examples,” in *POPL*, 2011, pp. 317–330.

[17] S. Chandra, E. Torlak, S. Barman, and R. Bodík, “Angelic Debugging,” in *ICSE*, 2011, pp. 121–130.

[18] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. D. Sutter, “Refactoring Using Type Constraints,” *ACM TOPLAS*, vol. 33, no. 3, 2011.

[19] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst, “Converting Java Programs to Use Generic Libraries,” in *OOPSLA*, 2004, pp. 15–34.

[20] F. Steimann and A. Thies, “From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility,” in *ECOOP*, 2009, pp. 419–443.