Auto-Locating and Fix-Propagating for HTML Validation Errors to PHP Server-side Code

Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
{hungnv,hoan,tung,tien}@iastate.edu

Abstract—Checking/correcting HTML validation errors in Web pages is helpful for Web developers in finding/fixing bugs. However, existing validating/fixing tools work well only on static HTML pages and do not help fix the corresponding server code if validation errors are found in HTML pages, due to several challenges with dynamically generated pages in Web development.

We propose PhpSync, a novel automatic locating/fixing tool for HTML validation errors in PHP-based Web applications. Given an HTML page produced by a server-side PHP program, PhpSync uses Tidy, an HTML validating/correcting tool to find the validation errors in that HTML page. If errors are detected, it leverages the fixes from Tidy in the given HTML page and propagates them to the corresponding location(s) in PHP code. Our core solutions include 1) a symbolic execution algorithm on the given PHP program to produce a single tree-based model, called D-model, which approximately represents its possible client page outputs, 2) an algorithm mapping any text in the given HTML page to the text(s) in the node(s) of the D-model and then to the PHP code, and 3) a fix-propagating algorithm from the fixes in the HTML page to the PHP code via the D-model and the mapping algorithm. Our empirical evaluation shows that on average, PhpSync achieves 96.7% accuracy in locating the corresponding locations in PHP code from client pages, and 95% accuracy in propagating the fixes to the server-side code.

Index Terms—Fix Propagation, Bug Localization, PHP Dynamic Web Applications, Validation Errors

I. INTRODUCTION

Web applications have become a critical infrastructure in our society. The World Wide Web Consortium (W3C) has developed several standards to ensure the development of high-quality and reliable Web applications [1]. An important quality criterion for a Web application is Markup Validity [2], which defines the validity of a Web document in HTML and other client-side markup Web languages according to their corresponding grammar, vocabulary, and syntactical rules.

Although modern Web browsers handle very well the parsing of even not well-formed HTML pages, some software defects in Web applications are not always easily caught due to the client-server and dynamic nature of Web contents. Checking HTML validation errors could really help the process of finding and fixing bugs in Web development. In a survey conducted by W3C [3], a majority of Web professionals stated that validation errors is the first thing they check whenever they run into a Web styling or scripting bug. Creating Web pages according to a widely accepted standard also makes them easier to maintain and evolve, even if the maintenance and evolution is performed by different developers [3].

Recognizing the importance of markup validity for Web pages, several organizations/individuals have produced automatic Web page validating tools (also called HTML validators). Some HTML validators (e.g. Tidy [4]) also provide automatic support for fixing markup errors to convert an HTML page into a well-formed one that conforms to HTML grammar and syntax. However, such auto-fixing tools work well only on static HTML pages and do not address several challenges in current Web development. The first challenge is that in a Web application, a client-side HTML page is often dynamically generated from the server-side code, which is written in different languages. For example, the server code is written in PHP, ASP, Perl, SQL, etc., while a client-side page is in HTML, JavaScript, CSS, and so on. The generated HTML code is embedded within the string literals or the values of variables in the server code. Moreover, those values are also scattered in multiple locations in server pages. For example, to produce an HTML table, multiple variables and string constants in different functions in the server code can be involved. Importantly, because the server code dynamically produces different client pages depending on run-time situations, if a validation error is found and reported in a Web page (e.g. via Tidy), it is challenging for its developers to manually map the buggy location(s) back to its source(s) in the server-side code.

We propose PhpSync, an auto-locating and fix-propagating tool for HTML validation errors in PHP-based Web applications. Given an HTML page produced by a PHP server page, PhpSync uses Tidy, an HTML validating/correcting tool to find any validation errors on the HTML page. If errors are detected, PhpSync leverages the fixes from Tidy in the given HTML page and propagates them to the corresponding location(s) in the PHP code. In the cases that Tidy cannot provide the fixes, the auto-locating function in PhpSync will help developers to quickly locate the corresponding buggy locations in PHP code from the buggy HTML locations found by Tidy. PhpSync does not require the input that produces the erroneous page.

The dynamic nature of a Web application is addressed via our symbolic execution algorithm that symbolically executes the given PHP program to create a single tree-based representation, called D-model, which approximates its possible HTML client page outputs. Each D-model represents a symbolic, string-based value that is resulted from the symbolic execution of any PHP expression(s). The D-model for the entire PHP server page or function is composed by the D-models resulted from the buggy location(s) back to its source(s) in the server-side code.
from the intermediate computations during the symbolic execution of the expressions in that page/function. Symbols in a D-model represent users’ inputs, data retrieved from databases, or unresolved values. A node in a D-model represents either 1) a determined value (e.g. a string literal), 2) a non-determined data value (e.g. a user’s input), 3) a concatenation operation, 4) a selection operation, or 5) a repetition operation on other nodes/values. This allows PhpSync to model the multi-valued and scattered server-side data and the multiple versions of client-side code generated from the server code.

Another fundamental technique in PhpSync is CSMap, an algorithm that maps any text in the given HTML page produced by the given PHP program to the corresponding PHP code location by mapping that text to the node(s) of the corresponding D-model. Then, our fix-propagating algorithm derives the fixing changes from Tidy to the given HTML page and propagates them to the locations in PHP via the established client-to-server mappings. CSMap is generic and can be used in other applications such as locating the corresponding buggy PHP places for other types of errors found in an HTML page.

Our empirical evaluation on real-world Web applications shows that PhpSync achieves on average 96.7% accuracy in locating the corresponding locations in PHP code from client pages, and 95% accuracy in fix-propagating to server code. The key contributions of this paper include:

1) PhpSync, an auto-locating and fix-propagating tool for HTML validation errors in PHP-based Web applications;
2) CSMap, a mapping algorithm from an HTML page (produced by a PHP page) to the corresponding PHP locations;
3) an empirical evaluation on several real-world Web applications to show PhpSync’s correctness and efficiency.

Section II presents a motivating example. Section III discusses our representation model. Associated algorithms are described in Sections IV and V. Section VI is for our evaluation. Related work is in Section VII. Conclusions appear last.

II. MOTIVATING EXAMPLE AND APPROACH OVERVIEW

This section presents an example that illustrates a bug caused by an HTML validation error and the challenges in fixing such errors in PHP-based Web applications.

A. An Example of a Bug on an Ill-formed Web Page

This example is inspired from an online social network system in which users are able to connect with peers/friends via posting and sharing news, pictures, and videos in their daily activities. In this system, a user can view and provide comments on the posts from his/her friends’ pages. Figure 1a displays such a page when a short news item on ASE 2011 is posted. Each post is followed by one or multiple comments and a textbox along with a submission button for a user to enter a new comment. After the comment is provided and the button is pressed, the new comment is expected to appear at the end of the comments’ list, and the textbox and the button would be positioned at the bottom of the page for another comment as in Figure 1b. However, when a user entered a comment, the textbox and the submission button appeared before that newly input comment (see Figure 1c). Assume that this bug was found and reported by a user on that page.

From the point of view of the developer of this Web-based social network application, in order to understand and fix this bug, (s)he would naturally first examine the HTML code of that Web page (Figure 2) to see if there was any error in its presentation structure. (S)He could do this verification manually or use an automatic HTML validator such as Tidy [4]. Assume that (s)he found that the code missed a closing tag </div> for the opening tag <div>: the last page division (lines 12-15) is included within the page division starting at line 9, making the textbox and button belong to the same page division for the comment list. When the user submitted a comment, it was appended to the end of the page division for comments and appeared below the textbox.

B. Challenges for Validation and Bug Fixing on Web Pages

The fix in the HTML code would be straightforward as the developer should add a </div> closing tag at line 11 for the corresponding open tag at line 9. However, that HTML page was dynamically generated from PHP-based server code (Figure 3). With a PHP-based Web application, (s)he must locate and fix the corresponding buggy code in the server. Doing this task manually is challenging in general due to several reasons.
1) The mapping/tracing from a client HTML page to server-side code is not straightforward. A Web application is a client-server one and generally developed in multiple languages. The server code could be written using a scripting language, e.g. PHP, while the client-side code is in HTML for presentation and JavaScript (JS) for data processing and event handling.

2) When the server-side code executes at the server, client-side code is generated and sent to a browser to execute there. That is, PHP-based server-side code dynamically produces different HTML pages depending on different inputs. For example, depending on the log-in information of a user at run-time, different files are included, different functions are executed, different execution paths in PHP code are taken in order to generate a particular client page. In this motivating example, to fix the bug, the developer would start examining the file `index.php` on the server side (Figure 3a) because (s)he found the error in the client page `index.php`. However, the bug is not within the file `index.php` of the server side. That PHP file is responsible for checking if a user has logged in (line 3, Figure 3a) via `is_logged_in` function in the file `functions.php` (line 2), which also contains other utility and formatting functions in the system (Figure 3d). The file `main.php` (Figure 3b) contains the code handling the cases of correct log-ins, while `error.php` (Figure 3c) handles the incorrect cases. In practice, validation errors are found in a client page via an HTML validation tool [4] and reported without corresponding input and action steps to produce that page. Thus, to fix them, a developer might have to check many server-side files and execution paths to find the right execution path that produces that client page.

3) Due to the dynamic nature of PHP/HTML/JS and the generation of client-side code, in a Web application code, and data tend to be mixed, especially client-side code is often embedded in server-side data. For example, the code of the `div` tags are embedded within PHP string literals. Moreover, a piece of HTML code might be generated via many PHP string literals, variables, and functions that are scattered in different places in the server-side code. In this example, the `<body>` element of the main HTML page is generated from the values of several scattered literals, variables, and function calls. To locate the right place to fix the `<div>` tag, the developer needs to check several literals and differentiate between many tags with the same name `<div>` that appear in several places in `main.php` and `functions.php`. In this example, the developer must determine that the error is in the `addComments` function in `functions.php` (line 12, Figure 3d). In reality, the numbers of included files, functions, variables, literals, and execution paths might be very high and they are scattered, thus making it challenging for a developer to manually locate the bug.

4) In this example, the PHP statement that prints out the erroneous HTML line is line 8 of Figure 3b: `echo addComments(...);`. However, to fix that error, a developer in fact must change line 12 of Figure 3d (`$output = "\n\n"`) where the erroneous HTML line is composed and manipulated.

This example shows that HTML validation errors could cause run-time bugs even when a browser can still display the page. As a user submitted a comment by clicking the button, the JS function comment (not shown) was invoked (line 11, Figure 3b). Due to the missing `<div>` tag, it incorrectly updated the corresponding division in the page via Ajax framework, thus, causing the incorrect page as in Figure 1c.

C. Approach Overview

We propose PhpSync, an auto-locating and fixing tool for validation errors in PHP-based Web applications. Given an HTML page produced by a server-side PHP program, PhpSync uses Tidy [4] to find the validation errors on the page. If errors are found, it propagates the fixes from Tidy on that HTML page to the corresponding location(s) in PHP code. The ideas are as follows: 1) PhpSync performs a symbolic execution to approximately represent all possible client-side HTML outputs of a server page \( S \) with a single tree-based model, called \( D-\)
A D-model is a tree-based representation for any symbolic, string-based value resulted from a symbolic execution on any portion of server-side PHP code. The D-model for the entire PHP server page/function is composed by the D-models resulted from the intermediate computations during a symbolic execution of the PHP expressions of that page/function. That is, PhpSync also creates D-models to represent possible values of intermediate computations and combines them into larger D-models for later computations. A D-model often contains symbols to represent user inputs, data retrieved from databases, or unresolved values. By performing a symbolic execution on a PHP page, PhpSync approximates all possible outputs/client pages with a single D-model. Let us explain it in details.

First, the string outputs for a portion of PHP code are stream-like, i.e. are produced via sequential writing or concatenation operations on PHP string values. The string value $T$ of a data-related PHP expression or the string value resulted from a string computation in PHP can be produced using the following context-free production rules [5]:

- **Rule 1**: $T \rightarrow t$
- **Rule 2**: $T \rightarrow T \cdot T$
- **Rule 3**: $T \rightarrow T \mid T$

Rule 1 says that, the value of a PHP expression can be a string literal. Rule 2 means that the value of a PHP expression can be concatenated from the values of two PHP expressions. Rule 3 specifies that a PHP expression can have either one of two values depending on the actual execution path at runtime. For example, in Figure 3a, the output of the page `index.php` is produced using Rule 3 due to the `if` statement at line 3 (i.e. it is either one of two strings), while the string output at line 9 of `main.php` is produced using Rule 2 (i.e. it is concatenated by four strings). Both production processes use Rule 1. Rules 2 and 3 are also used to repeatedly produce a value. For example, the value of variable `$output` of the function `addComments` in `functions.php` is produced by repeatedly using a `foreach` loop via Rule 2 (lines 10-11, Figure 3d). Those rules for output production of PHP code suggest the following structure.

**Definition 1**: A **D-model** is a labeled, ordered tree, in which the leaf nodes represent the values, and the inner-nodes represent the operations for combining those values.

1. There are two kinds of leaf nodes:
   - A **literal** node represents a determined string value (e.g. a PHP literal), and
   - A **symbolic** node represents an undetermined/unresolved string value (e.g. a user input).

2. There are three types of inner nodes, representing three kinds of operations on D-models:
   - A **Concat** node represents a value that is concatenated from the values corresponding to the sub-trees of that node. The order of the sub-trees represents the order of the concatenation operation.
   - A **Select** node represents a value that could be selected from the values corresponding to its sub-trees.
   - A **Repeat** node represents a value that could be repeatedly concatenated from the values corresponding to the sub-trees of the only child node of that **Repeat** node.

3. The nodes on D-models have their attributes describing additional information, such as the PHP expressions associated with literal and symbolic nodes.

Figure 4 illustrates a D-model that represents the output of the page `index.php` in Figure 3a. As seen, the root node of the D-model is a **Select** node, representing that the corresponding output of this PHP page is selected from the two values of two corresponding sub-trees of that root node. The left and right subtrees correspond to the outputs if `error.php` or `main.php` is executed, respectively. The root node of the right subtree is a **Concat** node representing the concatenation of the values of multiple literals (represented as literal nodes), e.g. the string literal “</div>” where the variables `$post` and `$comment` (represented as symbolic nodes), and the return values from different function calls. The return value of function call `addComments` is represented as the D-model rooted at the second **Concat** node, with its child node **Repeat** representing the repetition in the `foreach` loop. Consecutive string literals are combined for a compact D-model representation.

Note that a D-model approximates all possible symbolic outputs of PHP code by symbolically executing all of its execution paths. However, it does not represent all possible paths.

**B. Building D-model via Symbolic Execution**

We develop an algorithm to evaluate/compute the symbolic value for the output of any PHP code by building its D-model. It takes as an input the code of a PHP server page, and performs a symbolic execution to create a D-model for a special variable `$output`, to represent the output of that page. During execution, it creates the D-models for the intermediate results and updates the D-models for encountered variables.

The algorithm recursively evaluates all statements in all branches, updates/creates small D-models, and combines them into larger ones. It processes the PHP statements as follows:

1. $E \rightarrow \text{scalarValue}$: As a scalar/string value is encountered, a literal node is created to contain the corresponding string.
2. $E_1 \rightarrow \forall E_2$: Since a variable might have different values at different points in execution, PhpSync maintains for each variable $V$ a D-model corresponding to its most recent value during the execution. When meeting an assignment expression, PhpSync computes the D-model for the expression $E_2$, and assigns that D-model as the most recent value of $V$.
3. $E \rightarrow \forall V$: When a variable $V$ is retrieved for a computation, its latest D-model is used. However, if $V$ does not have any D-model, PhpSync returns a symbolic node representing an undetermined value. This corresponds to the cases of user inputs, data values from databases, or unresolved computations.
4. E1 → E2.E3: For an expression with a concatenation, PhpSync processes the sub-expressions to produce their D-models, and then creates the resulting D-model with its root node being a Concat node. The sub-trees of that root node are the computed D-models of the sub-expressions. Those subtrees are connected in the same order as the appearance order of the corresponding sub-expressions. PhpSync also performs other standard string and arithmetic operations in a similar process. Un-resolved results are represented as symbolic nodes.

5. S → echo E: When seeing an echo/print statement, PhpSync concatenates the current D-model of the variable $Output and the D-model of E to produce the new D-model for $Output. Note that $Output holds the current output of the PHP page.

6. S1 → if (E) S2 else S3: For an if statement, PhpSync executes both branches, and collects into a set $V*$ all variables $V$ modified in either branch. Let us use $V_{S2.D}$ and $V_{S3.D}$ to denote the D-models of $V$ after executing each branch, respectively. For each $V$ in $V*$, PhpSync updates its value with a new D-model. The new D-model is rooted at a new Select node whose children are $V_{S2.D}$ and $V_{S3.D}$. If the else branch is empty, the latest D-model for $V$ before the if statement is used in place of $V_{S3.D}$. The same treatment is for Switch statements.

7. S1 → while (E) S2: First, PhpSync executes statement S2 once and collects all modified variables $V$ into $V*$. Typically, the string value of a variable is appended during the execution of a loop. Let us use $D_{V}$ to denote the D-model that represents the symbolic string value appended to $V$. For a variable $V$ in $V*$, PhpSync updates its value with a new D-model. The new D-model is rooted at a new Concat node whose children are $VD$ and a new Repeat node (Figure 4). The Repeat node has $D_{V}$ as its only child. If the value of $V$ is not appended in the loop, PhpSync continues to try and retains the old value of $V$ before the loop. The same treatment is for a for statement.

8. S → return E: When PhpSync meets a return statement, the D-model of E is computed and collected into a set reValues of all possible returned values of the current function/file.

9. function call: When a function is called, PhpSync assigns the D-models of the actual arguments to the formal parameters of the function, and then performs a symbolic execution on the function’s code. After executing the function, it creates a new D-model with its root being a new Select node to describe the possibly multiple returned values of the function. The children of that Select node are the D-models in the reValues set of the function. If the function has only one returned value, the D-model of that returned value is used. If global variables and reference parameters are modified during the execution of the function, their D-models also updated accordingly. If the code of the called function is unavailable (e.g. library functions), it represents the returned value by a symbolic node.

10. E1 → include E2: PhpSync computes the string value from the D-model of E2 and considers it as a file name $fname$. Then, it continues the execution on that file. Finally, the D-
model of E1 is assigned with a new D-model whose root is at a new Select node with its children being all returned values after executing $\text{name}$ as in the case of a function call.

11. exit(); If PhpSync meets an exit function call, the D-model of $\text{Output}$ is collected into outputValues set of the current page.

12. block of statements: After executing all statements in the PHP program/page, PhpSync creates a new D-model with its root being a new Select node to describe the possibly multiple outputs of the page. The children of that Select node are the D-models in the set outputValues of the page.

While building the D-models, PhpSync also keeps the mapping between the D-model leaf nodes and their corresponding PHP fragments. For example, the literal node $\text{<div class=...>}$ under the lowest Concat node in Figure 4 is mapped to the fragment $\text{<div class=...>}$ on line 6 of Figure 3d. For the mapping of a symbolic node, PhpSync also keeps its execution trace. For example, the node $\text{$\text{post}$}$ of Figure 4 is mapped to line 4 of Figure 3d (inside the function’s body), and the trace includes line 4 of Figure 3d, line 6 of Figure 3b, and lines 2-3 of Figure 3b. That trace is useful for developers in examining the output corresponding to $\text{post}$ (i.e. line 7 of Figure 2).

The limitation of PhpSync lies in the approximation of the symbolic executions of if and for/while statements. The condition of an if is not evaluated and only string-appending operations on variables are handled in a loop. PhpSync also does not handle well library function calls if the source code is unavailable.

IV. CSMAP: MAPPING TEXTS OF CLIENT PAGE TO SERVER PAGE VIA D-MODEL

Let us present CSMap algorithm that maps any text in an HTML page to the corresponding location in a server page. It takes as inputs an D-model $D$ and a string $C$, divides $C$ into proper sub-strings and maps them to the corresponding literal or symbolic nodes in $D$, and then to PHP literals or variables.

A. Algorithm Design Strategies

A D-model $D$ for a server page can be considered as a context-free grammar (CFG) and a string $C$ is one of its concrete sentences. However, the traditional CFG parsing/compiling techniques [6] are not suitable and efficient here because the D-model always contains multiple symbols (i.e. symbolic values) that correspond to user inputs, etc. Therefore, we design CSMap with the following heuristic strategies:

1. Top-down and divide-and-conquer: with the goal of mapping texts to the leaf nodes in $D$, it is natural to perform the mapping of the substrings in $C$ to the sub-trees in $D$. CSMap follows the top-down process as in top-down parsers [6].

2. Pivoting: despite that the HTML pages are dynamically generated, the shared/static HTML code portions among (some of) those outputs of a PHP page occur very often. CSMap attempts to map the string $C$ to these shared code portions in $D$ first, and then uses them as the already-mapped pivots for further dividing and conquering. That is, the process will continue on the substrings of $C$ divided by those pivots.

3. Local best-matching: Since there may exist many selection nodes, CSMap could face the combinatorial explosion if it tries to exhaustively explore all combinations of their branches and perform optimal matching. Thus, for a selection node, CSMap uses a local best-matching strategy by first exploring all branches of the selection node and mapping to the branch with more matched characters. This choice is made locally for each selection without considering globally optimal matching.

B. Detailed Algorithm

Figure 5 shows the pseudo-code for CSMap algorithm. It is designed as the recursive function $\text{StrToDModel}$ whose inputs are a string $C$ and the root node $r$ of a D-model. There are five overloading functions $\text{StrToDModel}$ corresponding to five types of D-model nodes. During the execution, the attribute $\text{MapLocation}$ of each substring in $C$ is assigned with at most one reference to a node in the D-model (i.e. its mapped node). CSMap handles each of the five node types as follows:
1. If \( r \) is a literal node, \( r \) has a value \( \text{val} \). If \( \text{val} \) appears in \( \text{str} \) (i.e., is its substring), then the characters of that substring are mapped to \( r \). However, since \( \text{str} \) might have several occurrences of \( \text{val} \), by a greedy strategy, CSMap maps the first occurrence of \( \text{val} \) in \( \text{str} \) to \( r \), i.e., favoring the leftmost mapped string.

2. If \( r \) is a Concat node, CSMap considers \( \text{str} \) as a concatenation of the values corresponding to the sub-trees of \( r \). To find the optimal mapping, one might need to divide \( \text{str} \) into all possible sub-strings and map each of them to the corresponding sub-tree of \( r \). However, to simplify the divide-and-conquer step, CSMap uses the pivoting strategy. It finds a pivot by checking the string of a literal node among the sub-trees of \( r \) to see if it occurs only once in \( \text{str} \). If such a pivot exists, it is used to divide \( \text{str} \) into two sub-strings, and the list of child nodes of \( r \) into two sub-lists rooted at two new Concat nodes for further mapping (lines 16-19). If such a node does not exist, CSMap maps \( \text{str} \) to the first sub-tree of \( r \) and recursively maps the remaining texts in \( \text{str} \) (after the already-mapped portions) to the other sub-trees of \( r \) (lines 21-22).

3. If \( r \) is a symbolic node, CSMap checks whether the sibling node of \( r \) is a pivot. If it is, CSMap considers the string \( \text{str} \) as the value generated from \( r \), thus, maps all characters of \( \text{str} \) to \( r \). If a pivot does not exist, CSMap does not map \( \text{str} \) to \( r \) because it tries to map \( \text{str} \) with other sibling nodes of \( r \).

4. If \( r \) is a Select node, \( \text{str} \) is considered to be produced from one of the D-models corresponding to the sub-trees of \( r \). Thus, CSMap recursively maps \( \text{str} \) to each sub-tree of \( r \), and chooses the sub-tree with the higher number of mapped characters as the mapping for \( \text{str} \) (lines 43-45).

5. If \( r \) is a Repeat node, \( C \) is considered as the concatenation of the values produced by the sub-trees of \( D \) after some number of iterations. CSMap attempts to map \( \text{str} \) to the child node of \( r \), which represents the appendix string in one iteration. It will continue to map the remaining of \( \text{str} \) until no more mapping is gained (lines 52-53).

Finally, after determining the mapping between the client-page \( C \) and the D-model \( D \) via CSMap, PhpSync uses the mapping from \( D \) to PHP code established during building \( D \) to map the texts in \( C \) to PHP literals, variables, or statements.

**Example.** Let us revisit the example in Figure 2 with the D-model in Figure 4 to illustrate CSMap. CSMap starts by mapping the entire HTML page to the D-model rooted at a Select node. For a Select node, CSMap first attempts to map the code to each branch separately. In this case, the first branch is the string "... User not logged in...", which does not exist in the HTML code, thus it remains unmapped. The second branch, however, starts at a Concat node with several pivot nodes that are helpful for the mapping. In particular, its first, third, and fifth child nodes are string literals that occur exactly once in the HTML code, hence CSMap maps the corresponding substrings in the HTML code to those literal nodes.

The remaining sub-strings "ASE 2011<br>: Submission is now open." (line 7) and lines 9-10 of Figure 2, are mapped respectively to the remaining child nodes (i.e. the symbolic node $post and the next Concat node). For that Concat node, CSMap again finds that its first child node ("<div id='divComments...'\)) corresponding to line 9 (Figure 2) is a pivot. Therefore, it maps the remaining substring on line 10 to the Repeat node (Figure 4). For a Repeat node, its contained D-model rooted at the child node Concat is mapped to the substring repeatedly until no further mapping is found. In this example, the substring can be mapped to the two literal nodes and the symbolic node $comment after one iteration. Even if line 10 were repeated several times, CSMap would still map the text to the D-model with the Repeat node.

At this point, CSMap has evaluated both branches of the Select node at the root of the top-level D-model. Comparing the mapping results, it returns the mapping given by the second branch where all of the HTML code is successfully mapped. Since CSMap works heuristically, it is important that the mapping is done correctly in the top-level steps of the divide-and-conquer stack. A client page typically contains large chunks of texts that are likely to remain unchanged for different executions of the server page. This nature of client pages makes it likely for CSMap to find correct pivots in the early mappings. Incorrect mappings may occur at a later stage of the execution but produce less impact on the overall result since remaining texts to be mapped get much smaller.

V. Auto-Locating and Fix-Propagating to PHP

This section describes how PhpSync helps in auto-locating and fix-propagating for the validation errors to PHP code. The inputs include a given HTML page \( C \) produced by a PHP page \( S \). PhpSync uses Tidy [4], an HTML validator/corrector, to check \( C \) for HTML validation errors. If errors are found, it uses Tidy to produce the corrected version \( C' \) of \( C \).

**Auto-Locating.** There exist the cases in which Tidy is not able to provide the fixes [4]; however, it points out the buggy locations in the HTML page \( C \). In such cases, for each error location in \( C \), PhpSync uses CSMap to automatically locate the corresponding literal node(s) in the D-model of \( S \) and then locate the PHP literal(s) in \( S \). For example, via CSMap, the \(<\text{div}>\) opening tag on line 9 of Figure 2 is mapped to the first literal node of the second Concat in Figure 4, therefore, is correctly traced back to line 9 of Figure 3d.

**Fix-Propagating.** If Tidy can fix those errors, PhpSync will propagate those fixes through the mapping between \( S \) and \( C \) established by CSMap. Because Tidy does not provide the operations of the fixes but produces only the corrected version \( C' \), we developed CCMap algorithm to map the texts between \( C \) and \( C' \) to derive the fixing changes. The output of the algorithm is all the changes at the character level between \( C \) and \( C' \), which are then used to propagate to the server code. We design CCMap with three strategies:

1. **Token-based processing:** CCMap treats the client code \( C \) as a sequence of tokens, instead of syntactic units because \( C \) might not be fully parsable due to its validation errors.

2. **Divide-and-conquer:** Due to the nature of validation errors (missing closing tags, missing tag brackets, invalid tags, etc.), the fixes from Tidy leave the majority of \( C \) un-changed, i.e., \( C \) and \( C' \) share similar texts. CCMap maps the unchanged portions in \( C \) and \( C' \), and uses them as pivots as in CSMap.
Fig. 6. CCMap Algorithm: Deriving Fixes from Tidy

3. Similar-matching: to capture the replacement operations between $C$ and $C'$, we modify the standard longest common sub-sequence (LCS) to support the mapping of similar texts.

The pseudo-code of CCMap is in Figure 6. It first tokenizes $C$ and $C'$ into two sequences of tokens $T$ and $T'$, respectively (lines 2-3). Then, it uses the standard LCS algorithm LCS_Exact to find the pivot tokens (line 4). For each two successive mapped tokens $T[l]$ and $T'[r], l < r$, and their corresponding mapped tokens $T'[l']$ and $T'[r']$, it uses LCS_Sim to find the similar not-yet-mapped tokens in the aligned sequences $T[l+1..r-1]$ and $T'[l'+1..r'-1]$ (lines 5-7). LCS_Sim, at lines 13-20, is almost the same as the standard LCS except the way it compares the elements between two sequences (line 17). In LCS_Sim, two elements can be mapped if their string similarity exceeds a threshold $\sigma$. Function $\text{sim}(\cdot)$ measures the similarity of two strings by the ratio between the length of their LCS and their average length. For each mapped pair of token $T[i]$ and $T'[i']$ (both exact and similar ones), CCMap runs LCS_Exact on the two sequences of characters to map the corresponding characters (lines 8-9). To map all the characters in the code, it then translates the mapping results of the characters in the tokens in $T$ and $T'$ to the characters in $C$ and $C'$. It maps the previously-removed delimiters in $C$ and $C'$ using LCS_Exact taking already-mapped characters as pivots (lines 10-12).

All mapped characters are considered as unchanged. The un-mapped ones in $C$ and $C'$ are considered as deleted and added, respectively. Finally, from those derived changes to $C$, PhpSync finds the corresponding D-model’s literal nodes and then applies them to the corresponding PHP string literals in $S$. For example, PhpSync uses CCMap to detect that $</div>$ is added by Tidy at line 11 of Figure 2, i.e., inserted after the `\n` character between lines 10 and 11. That string is mapped by CSMap to the literal `\n` at line 12 of Figure 3d. Thus, PhpSync will make the change at line 12: `$\text{output} = \text{\n</div>}$;`

VI. EMPIRICAL EVALUATION

This section presents our empirical evaluation on PhpSync. Our research questions are 1) how accurately PhpSync maps HTML code to server code, and 2) how accurately it propagates the fixes from Tidy to server code. All experiments were carried out on a Windows 7 Home Premium 64-bit computer with CPU Intel Core i3-370M 2.40 GHz and 6GB RAM.

We collected six PHP systems from sourceforge.net in different sizes and domains (Table II). We read the code to gain the knowledge and set up those systems on our server with required databases and sample data. For each system, we selected multiple server pages for testing and built their D-models. Column ExFiles shows the average number of executed server files for a page. Columns Nodes and Control show the average number of all nodes and that of control nodes (Select/Repeat) in a D-model. Running time is in column Time.

A. Accuracy of Mapping Client Code and Server Code

To evaluate PhpSync’s accuracy in mapping the texts in HTML to PHP code, we first collected the HTML test pages from the subject systems by navigating through several HTML pages within that system on a Web browser. We recorded each page as an HTML test page by saving its corresponding HTML code and the navigation steps to get to that page (for later reproducing the page and checking). For each subject system, we selected the HTML pages with different presentations to have the samples of client pages with diverse page structures.
Our evaluation method is to use PhpSync to map every character in an HTML test page $C$ to the corresponding character in a PHP literal or PHP variable, and then to verify those mappings for all characters by the combination of a checking tool and human subjects. Remind that given an HTML test page, PhpSync divides its HTML contents into several text fragments and maps each fragment into the PHP literals/variables (Section 4). Because all of those fragments cover the entire HTML test page, to verify PhpSync’s mapping for each character, one can check the mapping for each of those fragments (called test fragments). The unmapped fragments are considered to have incorrect mappings.

To reduce the effort of manual verification from human subjects, we wrote an evaluation program that checks PhpSync’s mapping from every test fragment $f$ of the test page to a PHP literal $l$. If $f$ is mapped to a PHP variable, we examine the mapping manually. Otherwise, that program replaces only the first character $C$ in the literal $l$ in the PHP code $S$ with a special character (SC) that does not appear in the page $C$. We then executed the instrumented PHP code $S'$ and followed the same recorded navigation steps to produce the new HTML page $C'$. If in $C'$, the first character position in $f$ is replaced with that SC and all other positions in $C'$ are un-changed, we consider it as a correct mapping for that character. Moreover, in such a case of correct mapping for that character, if $f$ is exactly identical to $l$, we consider the mapping ($f \rightarrow l$) correct for all characters in the fragment $f$, and consider $f$ as a correctly mapped fragment. When other positions in $C'$ besides $f$ have been changed, the evaluation tool cannot conclude that the mapping is incorrect. For instance, there may exist a correct mapping from some client code to a PHP literal inside a for/while loop. When the client code $C'$ is produced, the SC character may appear multiple times in $C'$ due to the execution of the loop. Thus, in all other cases, we manually verified the mapping from $f$ to $l$ by understanding the program semantics.

In Table III, column Mapping shows the result on SchoolMate v1.5.4. We collected a total of 21 HTML test pages. In column Fragments, the sub-columns All, Auto, Man, and Corr. respectively show the number of all test fragments in the test page, the numbers of auto-evaluated, manually-evaluated, and correctly mapped fragments. In column Characters, the sub-columns All and Corr. show the numbers of all characters and correctly mapped ones in a test page. Acc. shows accuracy, i.e. the ratio of the number of correctly mapped characters over the total.

Column Mapping of Table IV shows the results for all subject systems. Processing time is in column Time. As seen, PhpSync achieves very high accuracy (an average of 96.7%) in character mapping with a small processing time (an average of 3 seconds for a test page of about 10,000 characters). Column Files shows us that on average a test page is produced by 6 PHP files. Thus, our tool could help reduce developers’ effort in finding the PHP locations for a given HTML text.

### B. Accuracy of Fix-Propagating to Server Code

We used the same set of HTML test pages in those systems for an experiment to evaluate PhpSync’s accuracy in fix propagation. For each test page $C$, we used Tidy to detect validation errors. If errors were found and Tidy was able to fix the page into $C^+$, PhpSync would be used to derive the fix-propagation results. As shown, PhpSync achieves high accuracy (an average of 95%) in fix propagation with small processing time. Importantly, it did not introduce any new validation error. The columns Fix-Propagating in Tables III and IV display the fix-propagation results. Columns Err., Tidy, and PS show the number of total HTML validation errors found by Tidy, that of errors fixed by Tidy, and that of errors fixed by PhpSync via fix-propagation. As shown, PhpSync achieves high accuracy (an average of 95%) in fix propagation with small processing time. Threats to Validity. Our experiments were on only 6 systems with 74 test pages. The selected systems and test pages might not be representative. However, the number of test fragments is very large (13,111), of which 3,550 were manually checked in 15 hours. During that process, human errors could occur. Currently, PhpSync does not completely handle object-oriented PHP, thus most of the selected systems do not contain many classes. Four out of six systems have only reasonable sizes and do not contain many loops for complex computational logics.
VII. RELATED WORK

Artzi et al. [7] introduced Apollo, a method to find bugs in Web applications by combining concrete and symbolic execution. It executes a Web application on an initial empty or randomly-chosen input. Additional inputs are derived by solving path constraints and conditions extracted from exercised control flow paths [7]. Failures during such executions are reported as bugs. In [8], they extended Apollo to also model interactive user inputs in a Web application. However, it does not pinpoint the buggy PHP statements that cause such errors.

To support such fault localization, in [9], they combined a variation of Tarantula [10] with the use of a dynamic output mapping technique. For each statement, Tarantula associates it with a suspiciousness rating that indicates the likelihood for the statement to contribute to a fault. The rating is computed based on the percentages of passing and failing tests that execute that statement. However, they reported that in a Web application, a significant number of statements/lines are executed in both cases, or only in failing executions. Thus, they combined Tarantula with a dynamic output mapping technique, which instruments a shadow interpreter to create a mapping between the lines in PHP and HTML code by recording the line number of the originating PHP statement whenever output is written out using the `echo` and `print` statements [9].

In comparison, while their output mapping technique is based on dynamic analysis with run-time instrumentation into an interpreter, PhpSync relies on symbolic execution. Their technique is lightweight, however, PhpSync is better suited for this auto-locating and fix-propagating problem. First, for an erroneous HTML line detected by an HTML validator, their tool will map it to the PHP statement responsible for printing it out. However, that PHP `print/echo` statement might not always be the line that needs to be fixed because the erroneous content of the HTML line might be composed and manipulated in string variables in previous statement(s) (see the motivating example in Section II.B.4). Second, in practice, validation errors could be found in a client page via Tidy and reported without corresponding input and action steps to produce that page. Thus, their dynamic mapping technique cannot be applied. In this case, a fixer can still use PhpSync to fix the errors. Finally, for fix-propagation, PhpSync performs mapping at the character level, while for the debugging purpose, their tool maps at the line level.

Tidy [4], an HTML validator/corrector, works mostly on static HTML pages. For PHP code, it filters all the code within a `<?php` and the corresponding `?>` and considers the remaining as HTML code. That scheme does not work well because HTML code is embedded within multiple scattered PHP literals and variables (see Section II). Similar to Tidy, other validating tools [11], [12], [2] are limited to support validating or correcting only client pages in XML/HTML/CSS.

Minamide’s string analyzer [5] takes a PHP program and a regular expression describing all of its possible inputs, and then statically approximates and validates the output via a context-free grammar. In comparison, his goal is to validate approximated HTML outputs from a PHP program without fixing support. Moreover, PhpSync performs symbolic execution requiring an input specification as in Minamide’s.

Using a string analyzer, Wang et al. [13] compute the approximated output of a PHP program and identify the constant strings visible from the browser for translation purpose.

Several string-taint analysis techniques were built for software security problems [14], [15], [16]. Gould et al. [17] use string analysis to guarantee well-typed SQL queries generated by a Java program. The type system in [18] is based on regular expressions with string concatenation and pattern matching. A CFG-based type system for string analysis is presented in [19]. PhpSync complements to PHP debuggers [20], however it does not need the inputs of PHP programs with symbolic execution.

VIII. CONCLUSIONS

We propose PhpSync, an auto-locating and fix-propagating tool for validation errors. Given an HTML page produced by PHP code, PhpSync uses Tidy to find its validation errors, and propagates Tidy’s fixes to PHP code. Our core solutions include a symbolic execution algorithm on PHP code to produce an D-model, which approximates all possible client pages, and the client-server mapping and fix-propagating algorithms. Our evaluation shows that it achieves high accuracy in both tasks.

ACKNOWLEDGMENTS

This project is funded in part by NSF CCF-1018600 grant. The first author was funded in part by a fellowship from Vietnamese Education Foundation (VEF).

REFERENCES