2016

Cross-language program analysis for dynamic web applications

Hung Viet Nguyen
Iowa State University

Cross-language program analysis for dynamic web applications

by

Hung Viet Nguyen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Suraj C. Kothari, Major Professor
Morris Chang
Manimaran Govindarasu
Daji Qiao
Samik Basu

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

This thesis would not be possible without the guidance and support of many people.

First and foremost, I am deeply grateful to my adviser, Prof. Suraj C. Kothari. I admire not only his knowledge and wisdom but also his warmth and kindness. He taught me the importance of selecting what problems to work on because "life is short," and made me question whether a method that I was about to use was going to achieve its goal. Much like the way that he leads a strong research group and company to deal with the complexity of software, he has guided me to handle life situations with grace, simplicity, and straightforwardness. He also encouraged me to help others without asking for anything in return. I feel very fortunate to have met Prof. Kothari in my life.

I am also grateful to Prof. Tien N. Nguyen, who had an important role in my professional development and made a large investment in my upbringing as a researcher. Without him, I would not have had a number of achievements during my Ph.D. study.

Next, I would like to sincerely thank Prof. Christian Kästner from Carnegie Mellon University, who is an excellent co-mentor for my research. I am often amazed by his sharp thinking and high productivity, among his many other exceptional traits. He set high standards for my research, but at the same time, gave me the skill set to succeed and instilled in me the passion for our work. I found my weekly meetings with him not only fun, productive, but also highly inspiring. It is truly a privilege and enjoyment for me to have worked with him.

For many years I have also benefited from the research group that I was a part of. It was perhaps the most productive and supportive lab environment that I have ever joined. This thesis is the fruit of working with my kind and talented colleagues.

My gratitude also extends to my committee members. Prof. Chang deserves special thanks since he was there when I needed help the most and gave me valuable advice for my future career as well as supporting my job applications. Prof. Govindarasu and Prof. Qiao not only helped with my Ph.D. program but also gave important guidance and encouragement. While my interaction with Prof. Basu was limited, I was already impressed with his openness and graciousness. I wish that I had spent more time with all the professors to learn more from them.

In addition, I am thankful to the members of the Vietnam Education Foundation Fellowship who provided financial support for the first two years of my study, to the chair and staff of the Electrical and Computer Engineering Department, as well as staff members of the Graduate College and the International Students and Scholars Office, who have facilitated my study and made my time at Iowa State University memorable.

Thanks to my amazing wife and my wonderful brother for always being by my side, to many dear friends for having been an essential part of my life outside the lab.

Finally, my deepest thanks and appreciation go to my Mom and Dad. For everything that you have done for me, there is certainly no way that I can thank you enough.

Overall, although bearing my name, this thesis is truly a collaborative effort of all of the aforementioned people. I also believe that getting this hard-earned degree is just the beginning—there is a lot more for me to explore and to contribute. As I continue my journey, I will remember those who have supported me during my years at Iowa State University. It has been a tough process, but it has also been a fun ride. Again, I am grateful to all of the people who made it happen. This thesis is dedicated to them.

ABSTRACT

Web applications have become one of the most important and prevalent types of software. In modern web applications, the display of any web page is usually an interplay of multiple languages and involves code execution at different locations (the server side, the database side, and the client side). These characteristics make it hard to write and maintain web applications. Much of the existing research and tool support often deals with one single language and therefore is still limited in addressing those challenges. To fill in this gap, this dissertation is aimed at developing *an infrastructure for cross-language program analysis for dynamic web applications* to support creating reliable and robust web applications with higher quality and lower costs. To reach that goal, we have developed the following research components. First, to understand the client-side code that is embedded in the server-side code, we develop an *output-oriented symbolic execution engine* that approximates all possible outputs of a server-side program. Second, we use variability-aware parsing, a technique recently developed for parsing conditional code in software product lines, to parse those outputs into a compact tree representation (called *VarDOM*) that represents all possible DOM variants of a web application. Third, we leverage the VarDOM to extract semantic information from the server-side code. Specifically, we develop novel concepts, techniques, and tools (1) to build *call graphs for embedded client code* in different languages, (2) to compute *cross-language program slices*, and (3) to compute a novel test coverage criterion called *output coverage* that aids testers in creating effective test suites for detecting output-related bugs. The results have been demonstrated in a wide range of applications for web programs such as *IDE services*, *fault localization*, *bug detection*, and *testing*.

CHAPTER 1. OVERVIEW

1.1 Software Development Support for Web Applications

Software has become a critical part of our society over the last few decades. Advances in computers and technology have impacted virtually every aspect of modern society including business, commerce, healthcare, transportation, education, science, technology, entertainment, and so on. Improving software quality and reducing the cost of software development have become increasingly vital goals since software failures and software malfunctioning are highly expensive. According to a study commissioned by the U.S. Department of Commerce's National Institute of Standards and Technology (NIST) in 2002 [110], the economic costs of faulty software in the U.S. are estimated to be in the range of tens of billions of dollars per year, representing nearly one percent of the nation's gross domestic product. To improve software quality and reliability, various methods have been introduced to aid developers in making software, for example, by providing automated support in modern integrated development environments (IDEs) for purposes such as bug detection, debugging, refactoring, code navigation, code completion, and many other tasks.

Despite their increased popularity in supporting traditional software applications, existing methods face fundamental barriers when being applied to *multilingual, dynamic web applications*, which have become one of the fastest growing and most important types of software in recent years [100], with the success of languages such as PHP and JavaScript (JS). As of February 2016, there are more than three billion Internet users and

nearly one billion websites (more than 53,000 of which are being hacked every day) [133]. While it is critical to facilitate the development of web applications and reduce the number of software defects, program analysis and IDE support for dynamic web applications are still highly limited compared to those for traditional software applications due to a number of challenges that dynamic web applications pose to web code analysis.

1.2 Challenges in Analyzing Dynamic Web Applications

Web applications follow the client-server model [149], in which the server and the client communicate over a computer network. The server serves as the service provider whereas the client serves as the service requestor. When the client sends a request to the server, the server-side program is executed and generates a response to the client in the form of client-side code written in HTML, JS, and CSS. At the client, a web browser executes the received client-side code to display a web page. Unlike *static* web applications in which the server-side program generates the same response to the client across different executions, an important characteristic of *dynamic* web applications is that, depending on different conditions in the server-side code, the server-side program may generate different *outputs* (i.e., different variants of client-side code). In this work, we are interested in *dynamic web applications* since they are the prevalent type of web applications [14]. Note that since the client-side code is *generated* from the server-side programs, analyzing a web application means analyzing server-side programs. As an example, Figure 1.1a displays a dynamic web application written in PHP, in which the server-side program generates three different outputs (Figure 1.1b).

Key challenges. The nature of dynamic web applications present two core challenges for program analysis. First, there exists *a mixture of server-side code and client-side code* within server-side programs. The client-side code often appears in server-side programs as scattered, incomplete string fragments. Many applications such as IDE ser-

(a) Server-side program in PHP, containing a mixture of server-side code and client-side code

```
1  mysqli_query($conn, "UPDATE Sessions SET active='1'...");
2  function createInput($name, $value) {
3      return '<input name="'.$name.'" value="'.$value.'" />';
4  }
5  $content = '';
6  if ($C1) {
7      $content = createInput('foo', '1');
8  }
9  else if ($C2) {
10     $content = createInput('foo', '2');
11 }
12 echo "<form>" . $content . "</form>";
13 echo "<script>validate(document.forms[0].foo);</script>";
```

(b) Client-side programs (output of server-side prorgram), containing only client-side code

Case 1: $C1 == TRUE

```
<form><input name="foo" value="1" /></form>
<script>validate(document.forms[0].foo);</script>
```

No error

Case 2: $C1 == FALSE && $C2 == TRUE

```
<form><input name="foo" value="2" /></form>
<script>validate(document.forms[0].foo);</script>
```

No error

Case 3: $C1 == FALSE && $C2 == FALSE

```
<form></form>
<script>validate(document.forms[0].foo);</script>
```

Error: undefined object document.forms[0].foo

(c) Output-oriented abstraction of the server-side program, containing only client-side code with #ifdefs

```
<form>
#if $C1
<input name="foo" value="1" />
#else
#if $C2
<input name="foo" value="2" />
#endif
#endif
</form>
<script>validate(document.forms[0].foo);</script>
```

Figure 1.1 An example PHP program generating three different outputs depending on some conditions

vices and bug detection would require an understanding of such fragmented client-side code while it is still embedded in server-side strings. Second, there could be an *exponential number of generated variants of client-side code* depending on the conditions in the server-side program. An analysis would need to first capture these large number of variations and then perform operations on them.

To illustrate these challenges, we discuss a specific type of analysis in web applications, namely detection of *undefined reference errors*. A reference to a program entity (e.g., a variable or a function call) is undefined at run time if the entity has not been declared in the current execution. These types of errors can lead to unexpected behavior of the program at run time, ranging from disruptive web service, blank pages, missing user information, unwanted error messages, to fatal crashes, input validation bypass, and

other security vulnerabilities [104]. In our example in Figure 1.1, in the first two cases (PHP variable $C1 evaluates to TRUE, or $C1 evaluates to FALSE and $C2 evaluates to TRUE), an HTML form with an HTML input named foo is generated. However, in the third case (both $C1 and $C2 evaluate to FALSE), no HTML input named foo is defined in the form. Therefore, when the JS code attempts to access the HTML input, an undefined variable error occurs. In this case, since the error occurs at a call to an input validation function, some malicious inputs could be injected into the web application.

Since undefined references cause the program to behave incorrectly and potentially present security threats, it is desirable to detect them early. Concretely, for our example, we aim to statically analyze the server-side program and detect the possible undefined reference error on line 13 of Figure 1.1a. To do that, the analysis would need to identify all possible declarations and references of client-side entities (HTML inputs and JS variables) and then match their conditions to detect a generated output variant in which a reference does not have a corresponding declaration. For instance, in Figure 1.1a, it would need to detect that the HTML input named foo is created on line 7 under condition $C1 and is created on line 10 under condition !$C1 && C2, whereas the JS reference document.forms[0].foo on line 13 exists in all cases (condition TRUE). By matching the constraints of the HTML input declarations and the JS reference, the analysis would then be able to report that the JS reference is undefined under condition !$C1 && !$C2.

Such analysis on the server-side programs is challenging due to the following unique characteristics of dynamic web applications:

1. Mixture of server-side code and client-side code. In essence, a dynamic web application is a program that generates another program. In the first stage, the server-side program is executed and uses server-side data to generate a client-side program by assembling string literals (e.g., HTML templates and JS functions) with custom computations. The client-side program is then executed by the client's web browser in the second stage. This dynamic code generation introduces three challenges:

- *Embedded client code:* The server-side code and the client-side code are intermixed within the same program. Code elements occurring in the generated client-side program are often *embedded* in string literals in the server-side program. For example, the opening and closing HTML <form> tags appear within the PHP string fragments on line 12 of Figure 1.1a. However, not all server-side strings contain client-side code—e.g., the string on line 1 contains an SQL query and is not related to the output. An analysis would need to understand the semantics of the string literals to analyze embedded client code.

- *Fragmented client code:* Often, the client-side code is *assembled from various sources* in the server-side source code. In our example, the HTML input foo does not appear verbatim in the server-side program but is instead created via a PHP function (lines 7 and 10); it is then concatenated with two PHP strings on line 12 at a different statement. The order in which HTML fragments are defined may be different than the order in which they appear in the output. For instance, the HTML input is defined before the HTML form but appears inside the form since it is printed after the <form> opening tag on line 12. In addition, each fragment of the embedded client code *may not form a valid syntactical unit* (e.g., the first string literal on line 3 contains an incomplete HTML input tag, and the name foo of the HTML input appears as an isolated string on lines 7 and 10). To recognize that the PHP string 'foo' contains the name of an HTML input, the analysis would need to detect that the function call createInput invokes the function defined on lines 2–4, which generates an HTML input with its name being the value of the function's first argument. A heuristic parser that scans for string patterns of client-side code would not work in such cases.

- *Cross-language interactions:* The source code of a dynamic web application is typically written in *multiple languages* for various tasks such as defining the application

logic, accessing a database, and displaying content. These languages include server-side ones such as PHP, JSP, ASP, SQL, and client-side ones such as HTML, JS, CSS. Data entities can have relations *across languages*, i.e. the value of a data entity computed in one language may affect the value of another entity in another language. In our undefined reference analysis, the declarations and references could also be cross-language (e.g., a JS variable referring to an HTML input). As another example, a program failure can occur in a language that is different than that of the root cause (i.e., the place where the error needs to be fixed). Thus, a technique to analyze dynamic web applications or to provide IDE services must be able to handle embedded code in multiple languages.

2. Exponential number of variants of client-side code. The same server-side program can generate different variants of client-side code depending on different user inputs, data from databases, and configuration options. A program of n sequential `if` statements can generate 2^n number of paths, potentially producing 2^n unique outputs. In our example, the HTML input `foo` is created under conditions `$C1` and `!$C1 && $C2`, but is not created in condition `!$C1 && !$C2`, resulting in an error in that case. During analysis, these conditions need to be taken into consideration to verify whether a property can hold across all possible variants. A naive brute-force approach that explores an exponential number of outputs individually would not scale.

For these reasons, it is nontrivial to analyze embedded client code while it is still embedded in the server-side programs. Existing analysis and contemporary IDEs support developers in writing and maintaining code either in the server-side code only or in the generated client-side code only. For example, the basic editor services in IDEs, such as syntax highlighting, syntax validation, auto-completion, "jump to declarations", and many others are standard for most languages, including PHP, but missing for the client-side code found in string literals of the server-side programs. While research on analyses of dynamic programming languages has advanced the ability of analysis support for

languages such as PHP and JS (e.g., [16, 44, 6]), existing approaches are still limited in addressing the mixture of server-side code and client-side code and dealing with a large number of variants of client-side code in dynamic web applications.

1.3 Key Ideas of Our Approach

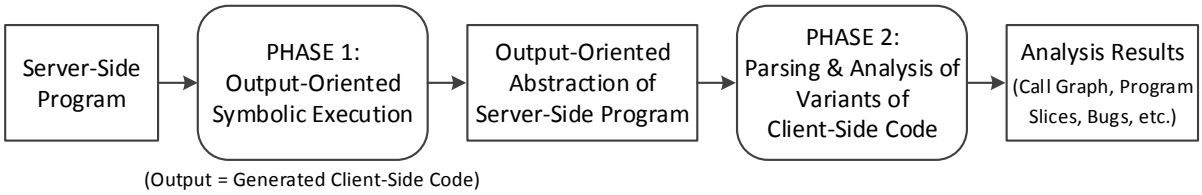


Figure 1.2 High-level overview of our approach

To tackle the above challenges, we have designed a two-phase approach to support web application analysis (see Figure 1.2). The two phases are aimed at addressing the first and the second challenge, respectively.

Phase 1—Output-oriented symbolic execution. The goal of this phase is to separate the client-side code from a mixture of server-side code and client-side code (challenge 1). To do that, we make use of symbolic execution [150] to abstract the server-side program with regard to its output. Symbolic execution explores different paths in a program and resolves possible values of a variable at a program point. (In a web application, the output stream can be considered as a special variable holding a string value representing the output.) Since we need to analyze the embedded client code of a web application, we use symbolic execution to resolve the string value of the output at the end of all possible executions. The result of this phase is an abstraction of the server-side program for its output, which contains purely client-side code and preserves possible client code variants controlled by the conditions in the server-side program. We retain the variants using the conditional compilation mechanism (#ifdefs) commonly used to specify a family of C programs. To illustrate, in Figure 1.1c, we show this abstraction

for our example. Note that we abstract away all parts of the server-side program that are not relevant to the output (e.g., the SQL query on line 1 of Figure 1.1a).

Phase 2—Parsing and analysis of variants of client-side code. The goal of this phase is to deal with variants of client-side code (challenge 2). Our key insight is that these variants resemble C code written with conditional directives (#ifdefs), except that it is for client-side code written in HTML, JS, and CSS. Our abstraction representation in phase 1 allows us to capture these variants and enables us to reuse and adapt the state-of-the-art approaches in the analysis for programs written C code with #ifdefs to apply for the analysis on client-side code variants written in HTML, JS, and CSS.

To illustrate the working of these two phases, we refer to our undefined reference analysis. After the first phase, we obtain an abstraction for the output of the server-side program (see Figure 1.1c). In the second phase, we use specialized parsers to parse the output abstraction and identify two HTML input declarations foo defined under conditions $C1 and !$C1 && $C2, and JS reference document.forms[0].foo under condition TRUE. We then check whether the reference is undefined under certain conditions. After matching the constraint of the reference against the combined constraints of all its declarations, the analysis can detect that the reference is undefined under condition !$C1 && !$C2.

By designing the two phases, we are able to tackle the core challenges in web application analysis. Phase 1 transforms a program with mixed server-side code and client-side code into one that contains only client-side code, and captures its variants via a representation resembling C code with #ifdefs. In phase 2, we develop specific techniques to parse and deal with client code variants. Different types of analysis can be performed not only for bug detection purposes as in this example but also for other purposes such as program understanding, programming support, or testing. The key limitations of our approach involve the inherent unsoundness of symbolic execution in the first phase and the possible combinatorial explosion of client-side code variants in the second phase. We discuss these limitations in detail at the end of Chapters 3 and 4.

1.4 Contributions and Outline of This Thesis

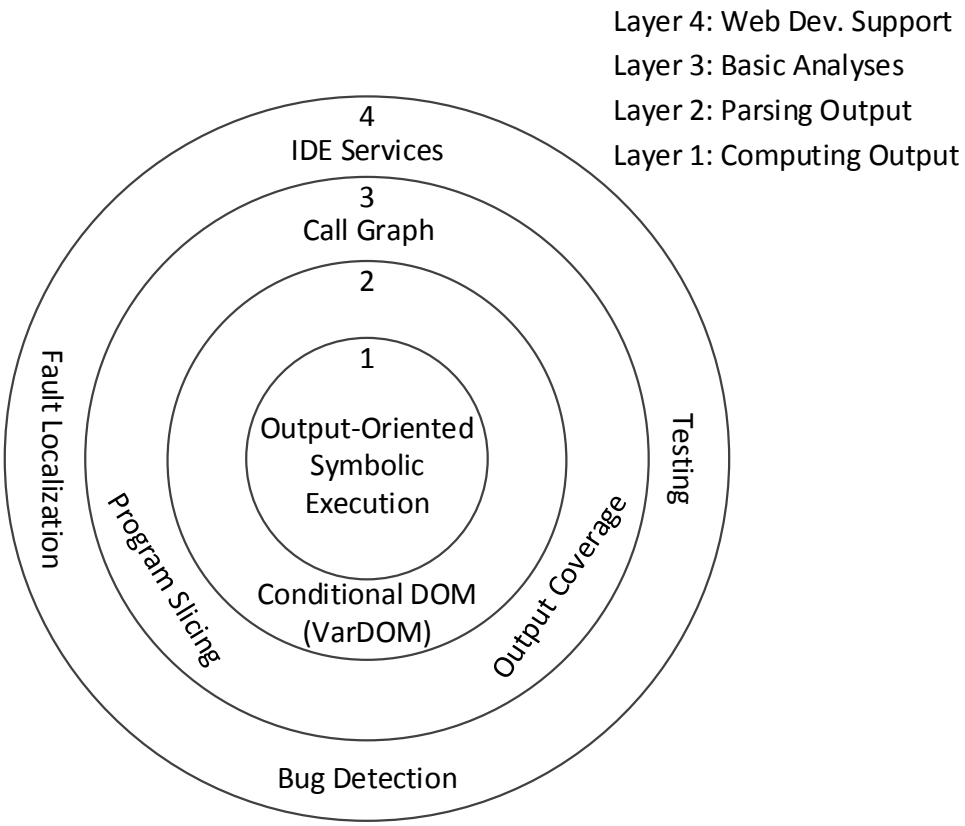


Figure 1.3 Overview of research tasks. Inner layers contain research components that those in the outer layers can be built upon.

Based on those key ideas, we propose *an infrastructure for cross-language program analysis for dynamic web applications*. As depicted in Figure 1.3, we organize the research tasks into four layers with components in the inner layers serving as the foundation for components in the outer layers to be built upon. (The first layer corresponds to the first phase, whereas the other layers correspond to the second phase.)

Layer 1—Computing the output of server-side code. We propose *output-oriented symbolic execution* [105, 102] on PHP code to approximate the output of a dynamic PHP web application. During symbolic execution, we follow all function calls, explore all branches, and keep track of the creation and propagation of string literals

and variables. We use symbolic values to represent unknown values (e.g., data from a database or web service, or the current time). The result of symbolic execution is the generated client-side code which possibly contains symbolic values and conditional values that are produced under specific path constraints.

Layer 2—Parsing the output of server-side code. Since the client-side code is conditional, we use variability-aware parsing [78] used in software product lines to parse the result from the previous step in order to understand the semantics of the client-side code. We create different variability-aware parsers for conditional HTML, JS, and CSS code. The parsing result is a *VarDOM representation* [101] of the embedded client code, which represents the hierarchical structure of a web page. A VarDOM, the core of our analysis framework, is similar to the Document Object Model (DOM) for HTML except that it contains condition nodes to indicate that certain subtrees of the HTML document may vary depending on some condition.

Layer 3—Providing basic cross-language analyses. Using the VarDOM, we are able to analyze the embedded client code written in HTML, JS, CSS. (While traditional analyses for static client code work on a single DOM, our variability-aware analyses work on a VarDOM with conditional parts.) We develop novel concepts, techniques, and tools for the following:

- For *call graphs in embedded client code* [101, 103], we create call-graph edges in different languages: between opening and corresponding closing HTML tags, between CSS rules and affected HTML elements, and between JS function calls and corresponding function declarations.

- For *cross-language program slicing* [102], we detect data flows within PHP and embedded languages (SQL, HTML, JS) and connect the data flows among them. Based on the established data flows, we compute a (possibly cross-language) program slice for any given entity.

- To support testing web applications, we introduce a novel test coverage criterion called *output coverage* and a technique for computing output coverage that measures how much of the output has been tested by a test suite.

Layer 4—Providing software development support. We use our program analysis infrastructure to support a wide range of dynamic web development activities. Our current support includes the following:

- *IDE services for embedded client code*: We develop *Varis* [101, 103] and *Babel-Ref* [106] to provide editor services on the client-side code of a PHP-based web application, while it is still embedded within server-side code. We implement various types of IDE services for embedded client code including syntax highlighting, code completion, "jump to declaration", and refactoring.

- *Fault localization via cross-language program slicing*: We implement *WebSlice* [102], a tool to compute cross-language program slices for PHP web applications. The program slices computed by WebSlice can be used to track data flows from the location that manifests the failure to the original location of the root cause for fault localization purposes.

- *Bug detection*: We develop two different tools for detecting two types of bugs in PHP web applications: (1) *Dangling Reference Checker (DRC)* [107, 104]—a tool that statically detects PHP and embedded dangling references, and (2) *PhpSync* [105]—an bug-locating and fix-propagating tool for HTML validation errors.

- *Output coverage visualization for output-oriented testing*: To guide testers in selecting additional test cases, we design a tool named *WebTest* that displays all possible outputs in one single web page and allows testers to visually explore covered and uncovered parts of all outputs. The testers can use WebTest to either augment

test cases or navigate and inspect the output directly to detect certain classes of presentation faults.

Besides the core analyses and services mentioned above, our framework is designed to be extensible and adaptable. The modules in layers 1 and 2 can be improved or replaced without breaking the working of the modules in the other layers. Layer 3 can be augmented with new type of analysis such as creating a cross-language program dependence graph for a web application. As for layer 4, web developers can not only make use of our existing tools but can also extend the framework to create new ones. For example, one can leverage the cross-language data flows and program slices provided in layer 3 to create a security tool that scans for sensitive flows and identifies vulnerabilities.

In summary, this thesis makes the following contributions:

1. **Concepts:** (1) the new concept of embedded client code in web code, (2) the notion of call graph for embedded client code, (3) the notion of program slices across different languages, and (4) the notion of different types of testers concerning different aspects of the software including its output, as well as a family of output coverage metrics of a test suite

2. **Representations:** The D-Model representation encoding different possible textual contents of the output and the VarDOM representation which compactly represents DOM variations generated from the server-side code

3. **Algorithms:** A systematic approach combining symbolic execution, variability-aware parsing, and variability-aware analysis (1) to build call graphs for embedded HTML, CSS, and JS code, (2) to compute cross-language data-flow relations and program slices, and (3) to compute output coverage metrics

4. **Tooling:** A toolchain that provides various kinds of software development support for dynamic web applications including IDE services, fault localization, bug detection, and testing

5. **Empirical studies:** Empirical evaluations on several real-world web applications to investigate the complexity of call graphs, data flows, program slices, the accuracy of bug detection, and the effectiveness of output coverage metrics, showing the analyses' accuracy, efficiency, and usefulness

The remainder of this thesis is organized as follows. Chapter 2 gives the necessary background on web application analysis, as well as some motivation as to why there is still a gap in supporting web development activities. Chapter 3–6 discuss in detail the techniques included in layers 1–4 of our framework, respectively. Next, Chapter 7 presents our empirical studies to investigate the accuracy, efficiency, and usefulness of our cross-language program analysis techniques. Finally, Chapter 8 summarizes the key ideas presented and make some concluding remarks.

CHAPTER 2. REVIEW OF LITERATURE

There exist a large number of program analysis approaches for web applications. However, existing work on dynamic web applications is still limited and lacks a foundation for program analysis methods as well as software development services for *multilingual web applications*. In the following, we discuss the literature based on different types of analysis for web programs and related techniques to the ones used in our framework.

2.1 Web Application Analysis

2.1.1 Analyzing the Output of a Web Application

Many researchers have investigated the output of web applications and the relationship between code and output for various purposes. Minamide [98] uses a string analyzer to statically approximate PHP-based, generated HTML pages and validate them. His string analyzer takes a PHP program and an input specification (in the form of a regular expression), which describes the set of possible inputs to the PHP program [98]. The analyzer approximates the output of a PHP program as a context-free grammar. Wang *et al.* [143] uses the string analyzer in Minamide's [98] to locate constant strings that need to be translated in a web application. It computes the approximated outputs of a PHP program and identifies the constant strings visible from the browser for translation via its flag propagation scheme.

2.1.2 Web Application Security

Several string taint-analysis techniques were built for PHP web programs and software-security problems [151, 153, 145]. Wassermann and Su introduced a string-taint analysis technique [144] based on Minamide's string analyzer in order to detect cross-site scripting security flaws [145]. Extending further, Wassermann *et al.* [146] developed an approach to generate test cases for such vulnerabilities. Also based on Minamide's work, several detection techniques were developed for SQL injection vulnerabilities in web scripts [151, 80]. Halfond and Orso [55] propose an approach to prevent SQL injection attacks. It uses program analysis to build a model of legitimate queries that can be generated from the application, and runtime monitoring for dynamic queries' inspection.

2.1.3 Bug Detection for Web Applications

Artzi *et al.* [17] 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 [17]. Failures during such executions are reported as bugs. In [18], 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 [16], they combined a variation of Tarantula [73] 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 [16].

Tidy [137], 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. Similar to Tidy, other validating tools are limited to support validating or correcting only client pages in XML/HTML/CSS.

2.1.4 Fault Localization for Web Applications

Clark *et al.* [30] support fault localization inside an SQL command embedded within JSP code. They cannot address the cases where the faults occurred in JSP and made the SQL command incorrect. Halfond *et al.* [90, 91] identifies the root cause of presentation failures in web applications that uses image processing and search-based techniques. There are several other *fault localization* approaches for individual language in traditional applications [73, 1, 74, 84, 126, 62, 36, 86, 88, 82, 2, 3, 115, 120, 154, 69]. However, they do not address the challenges in dynamic web applications.

2.1.5 IDE Services for Web Application Development

State-of-the-art IDEs do not provide call-graph-based editor support such as "jumps to declaration" for embedded client code. Recent works such as PHPQuickFix and PHPRepair [125] detect and fix errors in server-side PHP applications leading to ill-formed generated HTML. PHPQuickFix [125] examines *constant prints*, i.e., the PHP statements that print directly string literals and repairs HTML ill-formed errors. It analyzes each string literal separately and can only identify local issues. PHPRepair [125] follows a dynamic approach in which a given test suite is used to generate client-side code with different server-side executions, while tracing the origin of output strings.

2.1.6 Call Graph in Web Applications

There is a significant amount of work on constructing call graphs for JS code [43, 44, 54, 70, 71, 89, 129]. Accuracy and performance face challenges due to the dynamic nature of JS code and its interactions with the DOM and the browser. Kudzu [127] is a symbolic execution engine for JS. These tools are aimed at improving IDE services, but they all target plain client-side code, not code embedded in server-side programs.

2.1.7 Web Application Slicing

There has been rich literature in program slicing [87, 59, 24, 152, 21, 147, 82, 111, 109, 38, 48, 56, 113, 62, 28, 33, 45, 72, 60] and several excellent surveys on different techniques for program slicing [139, 22, 57, 58]. Harman *et al.* [57] provide an extensive survey with a classification with multiple dimensions in order to classify new program slicing techniques. Later, Silva extends the classification with a total of 12 dimensions [131]. None of existing program slicing approaches support cross-language slicing or PDG/CFG.

To approximate the dynamically generated client code, Tonella and Ricca [140, 122, 123] propose a flow analysis called *string-cat propagation* to associate the variables used in print/echo statements to string concatenations. They also combine with code extrusion, which unquotes the strings in echo. A slice is computed from such flows.

There also exists the information-flow approach to compute slices by Bergeretti and Carré (BC) [21]. The information-flow relations are recursively computed in a syntax-directed, bottom-up manner.

There are static slicing approaches based on various static analyses, e.g., incremental slicing [111], call-mark slicing [109], proposition-based slicing [38], stop-list slicing [48], amorphous slicing [56]. Our approach is related to PDG-based slicing [113, 62]; however, they do not deal with flows to embedded code. There are dynamic slicing approaches [82, 23, 72, 92], including language-independent slicing [23], which compute a slice for one specific execution and do not produce a static slice for all possible executions.

2.1.8 Web Application Testing

Supporting testing, especially web testing, from the perspective of a tester interested in the output has received increased attention recently.

First, Alshahwan proposed *output-uniqueness* test selection criterion [9, 8]. The criterion aims to maximize the differences among the observed outputs. They proposed seven syntactic abstractions pertinent to web applications to avoid sensitivity to nondeterministic output, and performed an empirical study to show that output uniqueness can be used as surrogate for whitebox testing. They measure coverage only as the absolute number of distinct observed outputs of a test suite, but have no notion of an output universe that could identify uncovered outputs.

Second, Zou *et al.* [155] introduce a V-DOM coverage for web applications. They convert a PHP program into C code and performs static analysis for control flows and data flows to build a V-DOM tree. V-DOM's nodes represent all possible DOM objects that can appear in any possible executions of a page. V-DOM coverage is defined as the ratio of the number of covered DOM objects over the total number of DOM objects.

Third, DomCovery [99] is another DOM-based coverage criteria for web applications. The coverage is defined at two levels: (1) the percentage of DOM states and transitions covered in the total state space, and (2) the percentage of elements covered in each particular DOM state.

With regard to output uniqueness for testing, several other researchers also introduced the concept of equivalent classes [112, 148, 53] where an element in a class leads to a correct output if all elements in the class lead to correct output. Subdomain partition methods are proposed for the input space via specification analysis [112]. Richardson and Clarke [124] use symbolic evaluation to partition the set of inputs into procedure subdomains so that the elements of each subdomain are processed uniformly by testing.

Overall, there is a rich literature on web testing [37, 83]. This adopts many quality assurance strategies developed in other contexts to the specifics of web applications,

including dynamic symbolic execution [146, 18, 127], search-based testing [7, 4, 93], mutation testing [117, 66], random testing [47, 15, 61], and model-based testing [121, 11]; they are all focused on analyzing the source code of the web application. Also, several specialized techniques to generate test cases by crawling the web page [94, 95, 51, 119, 96] and collecting session data [40] have been explored. As suggested previously [9], output coverage can be used a post-processing step to select a subset of the test cases generated by these tools in order to focus on output defects.

2.2 Related Techniques

2.2.1 Web Engineering

Web engineering is an important line of research that applies software engineering into web applications. Di Lucca *et al.* [34] developed Page Control Flow Graph (PCFG) to describe the dependencies among web pages. Earlier, they developed WARE [35], a dynamic analysis framework for reverse engineering of a web application. There exists much research on exploring flows among web pages for testing [15, 94, 128], for program comprehension [34], and for maintaining a distributed event-based system [49]. However, they do not build cross-language program slices at the level of entities. Orso *et al.* [6] use string analysis to discover client-side and server-side input validation inconsistencies.

2.2.2 String Analysis

There exist string analysis approaches for web programs and software security [98, 80, 145, 151, 153, 5]. They can be used to extract embedded code in our analysis. This line of research uses string analysis to approximate the output of server-side code. Minamide proposed a string analyzer [98] that takes a PHP program and a regular expression describing the input, and validates approximate HTML output via context-free grammar analysis. Wang *et al.* [143] compute the approximated output of PHP code and identify

the constant strings visible from a browser for translation. Both do not aim to analyze multiple variants of embedded code in JS or CSS. Several string taint-analysis techniques were built for PHP web programs and software-security problems [80, 145, 151, 153, 6].

In general, those approaches compute the output of the server code. However, they are limited in supporting the analysis of the semantics of the client code. We plan to provide foundational analyses that cross the client and server sides and multiple languages.

Outside the web context, there has been significant research on analyzing generators and guaranteeing invariants for the generated code [29, 63, 64, 108]. Staged programming languages such as MetaML integrate generators inside the language and can guarantee well-typedness of a program across all stages [75, 134]. They can give precise guarantees, but are only applicable with restricted, well-designed meta languages, not for arbitrary PHP/JS computations.

String embedding of DSLs is another form of two-stage computations [46], in which a string, such as an SQL command, is constructed in the host language and subsequently executed by a DSL interpreter. Here, the DSL code appears as string literals in a host language without IDE support, just as HTML code appears as strings in PHP. While other DSL-implementation approaches (e.g., pure embedded [65], extensible languages [41], external DSLs [79]) can potentially provide support for navigating DSL code, our infrastructure is applicable for simple string embedding as well.

2.2.3 Symbolic Execution

Symbolic execution was initially proposed as a program testing approach [81]. More recently, many forms of dynamic symbolic execution have been proposed to work around undecidability problems by combining symbolic execution with testing to guide the execution to parts of a program [26, 27, 52, 130]. We use a simple form of symbolic execution to explore many possible executions in a web application to produce possible outputs for further analysis.

2.2.4 Variability-Aware Parsing and Analysis

There is a community of researchers analyzing highly-configurable systems. The challenge of parsing all configurations of C code without preprocessing it first led to significant research on parsing, initially with restrictions or heuristics [20, 114] and then in a sound and complete fashion supporting arbitrary use of conditional compilation with our parser-combinator framework TypeChef [78] and later with a modified LR parser [50]. For an overview of this field, see a survey [135].

An abstract syntax tree with variations (either optional or choice nodes [42]) is a common abstraction for further variability-aware analysis over large configuration spaces [19, 25, 77, 85, 135], typically targeted at analyzing all configurations of a software product line without resorting to a brute-force approach of analyzing each configuration separately. Variability-aware analysis can be sound and complete with regard to analyzing all configurations separately. A common strategy to avoid reimplementing variability-aware versions of existing analyses is to reencode the build-time variability as runtime variability and use existing analysis mechanisms that can handle runtime variations, such as model checking [13, 116, 136].

Summary. Although there is a rich literature on web application analysis, existing works are still disconnected and are usually applied for a single language. They face fundamental barriers when dealing with the multilingual nature of dynamic web applications with embedded client code. Several techniques exist to target different aspects related to web applications such as string analysis, symbolic execution, and variability-aware parsing; however, they have not formed a systematic approach to target the core challenges of analyzing web applications. All of the above approaches either fail to address or only partially address the challenges in dynamic web applications discussed in Section 1.2. The goal of this thesis is to address that gap.

CHAPTER 3. OUTPUT-ORIENTED SYMBOLIC EXECUTION

This section explains our output-oriented symbolic execution for PHP web applications [105, 102]. To analyze a web application, both the server-side code and the generated client-side code need to be examined. Therefore, our goal is to use symbolic execution to identify how the client-side code will be generated in all possible executions of the server-side generator. We store the result of symbolic execution with a representation that we call D-Model [105], which compactly represents different possible variants of the output (or generated client-side code). Let us first explain the D-Model representation and then describe our symbolic execution to generate a D-Model.

3.1 D-Model: Representation of Outputs

Definition 1 *A **D-Model** is an ordered tree, in which the leaf nodes represent concrete or symbolic values, and the inner nodes represent the computations on those values.*

There are two kinds of leaf nodes:

- *A **Literal** node represents a concrete string value (e.g., "Welcome").*
- *A **Symbolic** node represents an unresolved string value (e.g., data from user input).*

There are two kinds of inner nodes:

- *A **Concat** node represents a value that is concatenated from the values corresponding to the sub-trees under 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 depending a condition.*

A node on a D-Model also contains information about its associated PHP expression and its original location in the server-side code (e.g., a Literal *node is associated with a PHP string constant in the source code, and a* Select *node is associated with the PHP expression for its condition).*

D-Model is used to represent any value resulted from a symbolic execution on any portion of the server-side PHP code (it can also be used for other server-side languages such as JSP and ASP). The D-Model for an entire PHP page is composed by the D-Models resulted from the intermediate computations during a symbolic execution of the PHP expressions of that page. That is, our approach creates D-Models to represent possible values of intermediate computations and combines them into larger D-Models for later computations. We also track the origin locations of all string values. By performing symbolic execution on a PHP page, our approach approximates all possible outputs/client-side pages with a single D-Model.

For illustration, we use a slightly larger example than the one in the first chapter to illustrate our concepts. Figure 3.1 shows an example PHP web application adapted from AddressBook-6.2.12. The main program (Figure 3.1a) generates different HTML input fields in an HTML form and different definitions of two JS functions with the same name update, depending on whether the AJAX option is enabled. The top and bottom parts of the page are generated by the files in Figures 3.1b and 3.1c, respectively. Note that the client code (written in HTML, JS, CSS) often appears in the server code as string literals or inline code (which is separated from PHP code by the directive <?php...?> and is sent verbatim to the client side when the PHP program is executed).

Figure 3.2 displays a D-Model that represents the output of the index.php page. The root node of the D-Model is a Concat node, representing that the corresponding output of

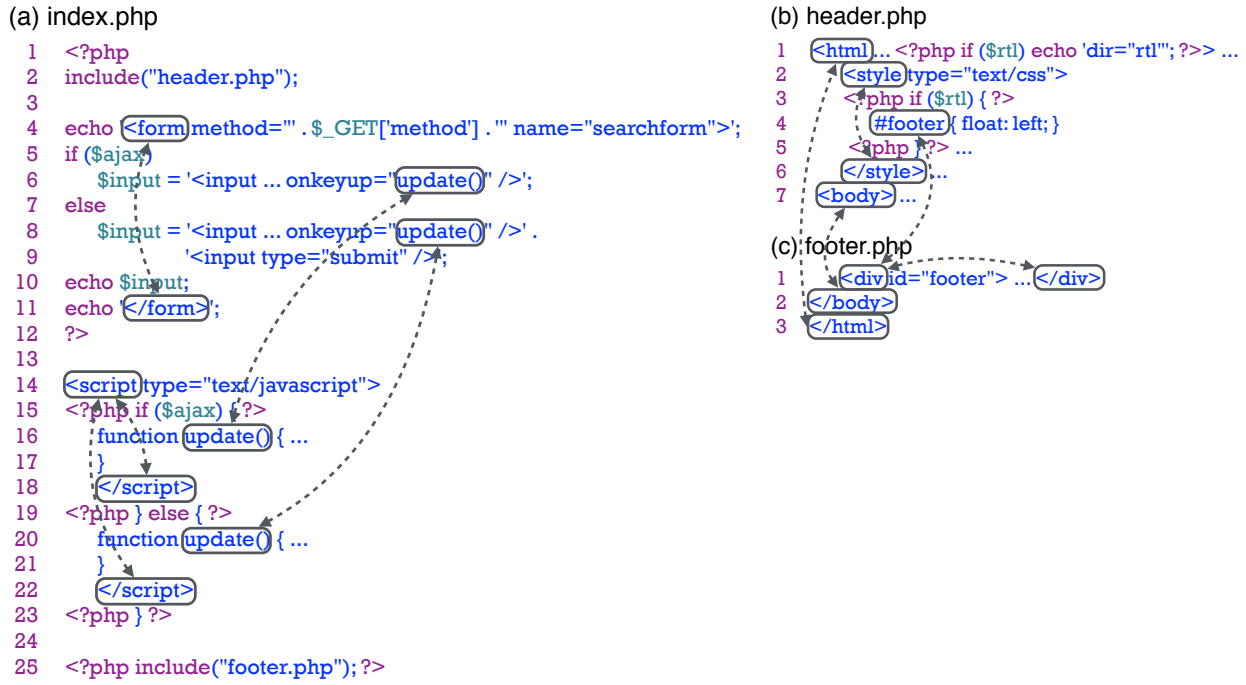


Figure 3.1 An example PHP program with call-graph edges for embedded client code (string literals and inline client code are highlighted in blue)

this PHP page is concatenated from multiple values. Some of the values can be selected from alternative values, which are represented by the sub-trees of a Select node. For example, the left and right sub-trees under the Select node α ($rtl) represent the string literal dir="rtl" and an empty string, corresponding to the two cases that the configuration option $rtl evaluates to TRUE or not. We use Greek letters to represent symbolic values in the conditions of Select nodes or unknown values in Symbolic nodes (e.g., $_GET['method']).

3.2 Symbolic Execution to Approximate Outputs

3.2.1 Key Ideas

We develop a symbolic execution engine called *PhpSync* to approximate all possible alternatives of the output of a PHP web application. Our key insight is that although the number of concrete outputs may be infinite, there are usually a finite number of structures

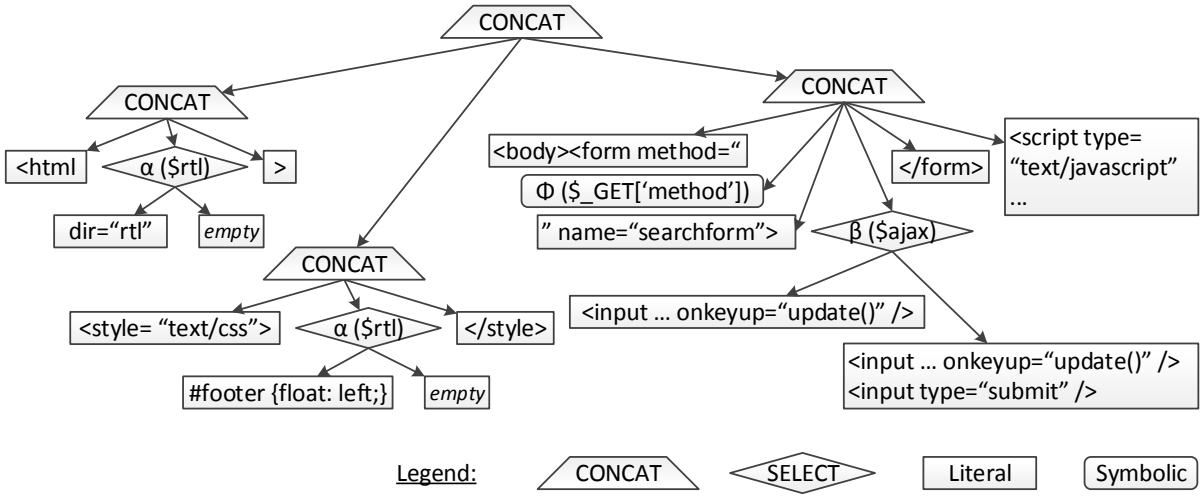


Figure 3.2 D-Model representation for the output of the PHP program in Figure 3.1

on a web page that we can derive. During execution, PhpSync considers all unknown values, such as user input and data from a database, as *symbolic values* (represented by D-Model Symbolic nodes). When reaching control-flow decisions, it explores all possible paths, keeping track of the current path condition and ignores executions with infeasible path constraints. Note that output fragments can be concrete, produced from string literals inside the PHP code (possibly after several reassignments, concatenation, and other string processing steps), or symbolic. During symbolic execution, we track all output of the executed PHP code and record the path constraint under which each output fragment was produced. Additionally, we track the origin location of each string literal such that we can map all output back to the original PHP literals. The result of symbolic execution is a D-Model representing the number of possible client-side implementations generated from the PHP code, in which each character or symbolic value has a path constraint and origin information. Our symbolic executor processes a PHP page using specific rules and repeats this process on other pages to approximate the output of all pages in a web application.

3.2.2 Evaluation Rules

We use the following notation to describe our technique. V is the set of all values (including special values represented by D-Models). C is the set of all *control* codes that represents the returned values of statements (e.g., 'RETURN' or 'BREAK'). S, E, and N are the sets of all statements, expressions, and identifiers, respectively. Π is the set of all path constraints; each constraint is a propositional formula. $\mathcal{P}(S)$ is the powerset of a set S. We use small letters for elements of a set (e.g., $s \in S$ is a statement).

Figure 3.3 shows the key evaluation rules. During symbolic execution, we maintain a program state $(\mathcal{V}, \pi)$ where the *value store* $\mathcal{V} : N \mapsto V$ is a (total) function mapping a variable/function name to its value (uninitialized variables have a $\perp$ value), and the *path constraint* π encodes the branch decisions taken to reach the current state. For a statement s, a rule $\langle\langle s, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle c, \mathcal{V}', \pi'\rangle\rangle$ denotes that the execution of s changes the program state from $(\mathcal{V}, \pi)$ to $(\mathcal{V}', \pi')$. The returned value c is a *control* code: It returns 'OK' if there was no control-flow breaking instruction in s (i.e., the next sequential statement can be executed) and other control codes (e.g., 'RETURN') otherwise. For an expression e, a rule $\langle e, \mathcal{V}, \pi\rangle \rightarrow \langle v, \mathcal{V}', \pi'\rangle$ denotes that the evaluation of e results in a new program state and returns a (non-control) value v. We use *addOutput* to record a string or symbolic value in the output (under a path constraint). Other notation and auxiliary functions are listed at the end of Figure 3.3. PhpSync processes PHP statements and expressions as follows.

Variable access (rule 1). When a variable is accessed for a computation, PhpSync looks up its value in the value store $\mathcal{V}$.

Assignments (rule 2). PhpSync computes the value of the right-hand-side expression and updates the value store $\mathcal{V}$ with this new value of the variable on the left-hand side of the assignment. The computed value is represented by a D-Model (or a Literal node if the right-hand-side expression is a string literal).

Initialization:

$$\mathcal{V}(x) = \bot \quad \pi = \mathsf{TRUE}$$

1. Variable Access:

$$\frac{v = \mathcal{V}(n)}{\langle \$n, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}, \pi \rangle}$$

2. Assignment:

$$\frac{\langle e, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \pi \rangle}{\langle \$n = e, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}'[n \mapsto v], \pi \rangle}$$

3. If Statement:

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad isSat(\pi \wedge \pi') \quad isSat(\pi \wedge \neg\pi') \\ \langle\!\langle s_1, \mathcal{V}', \pi \wedge \pi' \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \pi \wedge \pi' \rangle\!\rangle \quad \langle\!\langle s_2, \mathcal{V}', \pi \wedge \neg\pi' \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \pi \wedge \neg\pi' \rangle\!\rangle \\ \mathcal{V}_3(x) = select(\pi', \mathcal{V}_1(x), \mathcal{V}_2(x))\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \pi \rangle\!\rangle \rightarrow \langle\!\langle select(\pi', c_1, c_2), \mathcal{V}_3, \pi \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad \neg isSat(\pi \wedge \neg\pi') \\ \langle\!\langle s_1, \mathcal{V}', \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \pi \rangle\!\rangle\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \pi \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad \neg isSat(\pi \wedge \pi') \\ \langle\!\langle s_2, \mathcal{V}', \pi \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \pi \rangle\!\rangle\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \pi \rangle\!\rangle}$$

4. Function Declaration:

$$\frac{\lambda \text{ is a pointer to function } n(\$n_1, ..., \$n_m)\{s\}}{\langle\!\langle \mathsf{function}\ n(\$n_1, ..., \$n_m)\{s\}, \mathcal{V}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathsf{OK}, \mathcal{V}[n \mapsto \lambda], \pi \rangle\!\rangle}$$

5. Function Invocation:

$$\frac{\begin{array}{c}\lambda = \mathcal{V}_0(n) \quad \lambda \text{ is a pointer to function } n(\$n_1, ..., \$n_m)\{s\} \\ \langle e_i, \mathcal{V}_{i-1}, \pi \rangle \rightarrow \langle v_i, \mathcal{V}_i, \pi \rangle,\ \forall i \in [1..m] \quad \mathcal{V}_f(x) = \begin{cases} v_i & \text{if } x = n_i \\ \bot & \text{otherwise} \end{cases} \\ \langle\!\langle s, \mathcal{V}_f, \pi \rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}_{f'}, \pi \rangle\!\rangle\end{array}}{\langle n(e_1, ..., e_m), \mathcal{V}_0, \pi \rangle \rightarrow \langle \mathcal{V}_{f'}(\text{`RET'}), \mathcal{V}_m, \pi \rangle}$$

6. Return Statement:

$$\frac{\langle e, \mathcal{V}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \pi \rangle}{\langle\!\langle \mathsf{return}\ e, \mathcal{V}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathsf{RETURN}, \mathcal{V}'[\text{`RET'} \mapsto v], \pi \rangle\!\rangle}$$

Figure 3.3 Evaluation rules for output-oriented symbolic execution

7. Block of Statements:

$$\frac{\langle\langle s_1, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle c_1, \mathcal{V}_1, \pi\rangle\rangle \quad \pi' = whenEqual(c_1, \mathsf{OK}) \quad isSat(\pi \wedge \pi') \quad \langle\langle s_2, \mathcal{V}_1, \pi \wedge \pi'\rangle\rangle \rightarrow \langle\langle c_2, \mathcal{V}_2, \pi \wedge \pi'\rangle\rangle \quad \mathcal{V}_3(x) = select(\pi', \mathcal{V}_2(x), \mathcal{V}_1(x))}{\langle\langle s_1 s_2, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle select(\pi', c_2, c_1), \mathcal{V}_3, \pi\rangle\rangle}$$

8. While Statement:

$$\frac{\langle\langle \mathsf{if}\ (e)\ \{s\ \mathsf{while}\ (e)\ s\}, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle c, \mathcal{V}', \pi\rangle\rangle}{\langle\langle \mathsf{while}\ (e)\ s, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle c, \mathcal{V}', \pi\rangle\rangle}$$

9. Include Expression:

$$\frac{\langle e, \mathcal{V}, \pi\rangle \rightarrow \langle v, \mathcal{V}_1, \pi\rangle \quad \langle\langle s, \mathcal{V}_1, \pi\rangle\rangle \rightarrow \langle\langle c, \mathcal{V}_2, \pi\rangle\rangle \quad s = \begin{cases} parseFile(v) & \text{if } v \text{ is a concrete value} \\ \text{empty statement} & \text{otherwise} \end{cases}}{\langle\langle \mathsf{include}\ e, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle \mathcal{V}_2(\text{`RET'}), \mathcal{V}_2, \pi\rangle\rangle}$$

10. Infix Expression:

$$\frac{\langle e_1, \mathcal{V}, \pi\rangle \rightarrow \langle v_1, \mathcal{V}_1, \pi\rangle \quad \langle e_2, \mathcal{V}_1, \pi\rangle \rightarrow \langle v_2, \mathcal{V}_2, \pi\rangle \quad v = \begin{cases} concat(v_1, v_2) & \text{if } op \text{ is concatenation} \\ symbolic(e_1\ op\ e_2) & \text{otherwise} \end{cases}}{\langle e_1\ op\ e_2, \mathcal{V}, \pi\rangle \rightarrow \langle v, \mathcal{V}_2, \pi\rangle}$$

11. Echo Statement:

$$\frac{\langle e, \mathcal{V}, \pi\rangle \rightarrow \langle v, \mathcal{V}', \pi\rangle \quad addOutput(v, \pi)}{\langle\langle \mathsf{echo}\ e, \mathcal{V}, \pi\rangle\rangle \rightarrow \langle\langle \mathsf{OK}, \mathcal{V}', \pi\rangle\rangle}$$

Notation and auxiliary functions:
- $\mapsto$ denotes total functions.
- $g = f[x \mapsto y]$ denotes a function same as f except that $g(x) = y$.
- $symbolic(e)$ returns a fresh symbolic value mapped to an expression e.
- $select(\pi, v_1, v_2)$ returns an alternative value of v_1 or v_2 depending on π.
- $concat(v_1, v_2)$ returns a concatenation of v_1 and v_2.
- $isSat(\pi)$ returns TRUE if π is satisfiable and FALSE otherwise.
- $whenEqual(v, v')$ returns the constraint under which v equals v', e.g.
$whenEqual(select(\alpha, \mathsf{TRUE}, \mathsf{FALSE}), \mathsf{TRUE})$ returns α,
$whenEqual(select(\alpha > 1, \mathsf{TRUE}, \mathsf{FALSE}), \mathsf{TRUE})$ returns (fresh) β.
- $parseFile(v)$ parses a PHP file v and returns the parsed program.
- $addOutput(v, \pi)$ records value v under constraint π in the output.

Figure 3.3 (Continued)

Symbolic execution rules on PHP code to build D-Models

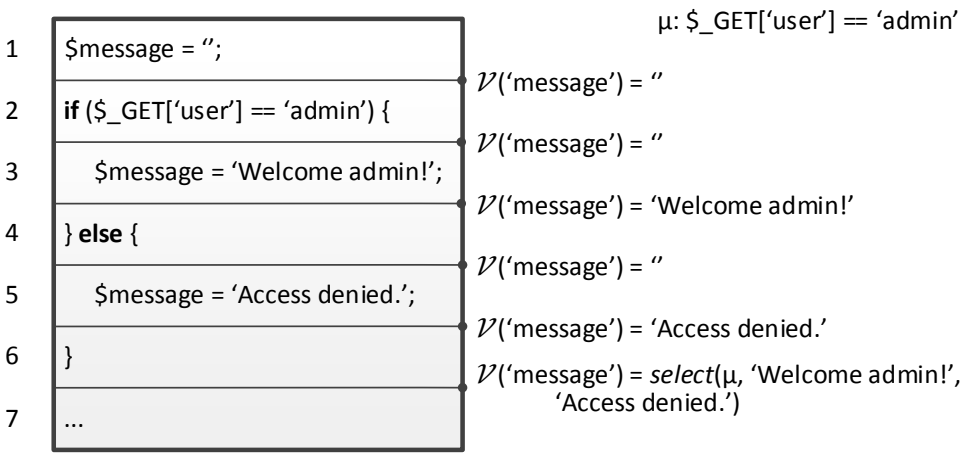


Figure 3.4 Executing conditional statements

Conditional statements (rule 3). If the path constraints of both branches of an if statement are satisfiable, we explore both branches. The function $whenEqual(v, \mathsf{TRUE})$ is used to compute the constraint where a value v (evaluated from the if condition) evaluates to TRUE. For example, the conditions on lines 2 and 7 of Figure 3.4 are both resolved into α == 'admin' where α is the symbolic value for $_GET['user']; thus, we evaluate the both conditions into the same (fresh) symbolic value μ (to simplify constraint checking). Modifications to the value store $\mathcal{V}$ take effect in the corresponding branch only. After executing the branches, we update the value store with the combined values from the two branches together with their corresponding constraints. To represent that a variable may have multiple values depending on a path constraint, we use a D-Model Select node ($select(\pi, v_1, v_2)$) to represent a selection between value v_1 if the path constraint π evaluates to TRUE and v_2 otherwise (we also use Select for control codes). Note that if the path constraint of one of the branches is unsatisfiable, we execute the other (satisfiable) branch only. As an illustration, in Figure 3.4, the variable $message after line 6 has two alternative values from both branches.

Functions (rules 4–6). Similar to concrete execution, PhpSync evaluates a function call in three steps (if the source code is available; otherwise, it returns a symbolic

value): First, it sets up a new context/call stack for the function and passes the actual parameters to the formal parameters of the function. Second, it executes the function body and records all returned values (here represented by a special variable named 'RET') encountered when exploring different paths in the function. Third, the returned value(s) are propagated to the call site of the function. The details are shown in rules 4–6. Note that for scalability, PhpSync does not execute a recursive function call.

Block of statements (rule 7). In a block of statements, the returned control code after executing a statement can be 'OK', indicating that the next statement can be executed, or other control codes otherwise (e.g., 'RETURN' for a return statement). Note that the returned control code can also represented by a D-Model Select node (e.g., the returned code of an if statement). Therefore, after each statement, we compute the path constraint under which the next statements can be executed (i.e., the constraint with which the returned control code equals 'OK') and execute them under that restricted constraint. After executing the block, we update the value store similarly to the case of an if statement. If the computed constraint is not satisfiable, we simply stop the execution for that block (not shown). In rule 7, we show the algorithm for a block of two statements. Note that the rule for a block with more statements can be generalized from this rule.

Loops (rule 8). We execute a loop by modeling the loop as recursively nested if statements with the same condition and body code. We first applying the rule for an if statement to execute the first iteration and then recursively execute the next iterations in the same manner. If the loop contains control-flow breaking instructions (such as break, continue, or exit), we either abort the loop (for break, return, and exit) or continue the next iteration (for continue) in their respective constraints (not shown). For scalability, we typically limit the number of iterations at one (i.e., the loop terminates after at most one iteration).

Dynamically included files (rule 9). A PHP program can dynamically include other files. During symbolic execution, we execute these files if the file names can be resolved to concrete values. Since include is an expression in PHP, we treat the returned value of include similarly to the returned value of a function call.

Computing the output (rules 10–11). The output of a PHP program is usually a concatenation of multiple string values and is printed out through echo/print statements or inline HTML code. To keep track of concatenations, we use a D-Model Concat node ($concat(v_1, v_2)$) to represent a concatenation of two (possibly symbolic) values v_1 and v_2 (rule 10). At echo/print statements or inline HTML code, we simply record the computed value v of expression e for the output in the corresponding path constraint (rule 11). (An echo e statement is a concatenation of the value of e with the current output, i.e. it can be treated as an equivalent assignment \$OUTPUT = \$OUTPUT . e, where \$OUTPUT is a special variable representing the current output.) The use of Concat values and Select values allows us to track the symbolic output with conditional fragments efficiently.

Limitations. We have made several design strategies to our symbolic-execution engine so that it can scale when computing all possible outputs of a PHP web application. Because of these simplifications, the engine has several limitations. Currently, we handle common PHP constructs and functions only since the PHP APIs are large. Specifically, (1) we have limited support for control-breaking statements (e.g., BREAK, RETURN, EXIT) or operations with objects and arrays in the presence of symbolic or conditional values, and (2) we implement infix expressions with the concatenation operator only since we are interested in the string output of a program (for other operators, we create fresh symbolic values to represent the results—for instance, we track $\alpha > 1$ as a new symbolic value β). Because of the conservative approximations with symbolic values and the limitations of external constraint solvers (especially with strings and objects/arrays), the engine may explore some infeasible paths. It runs exactly one iteration of each loop and skips

recursive function calls. Our approach also does not handle well library function calls in the presence of symbolic and conditional values since their source code is unavailable. Similarly, if the currently executed file invokes another file (by using the PHP expression include) and the expression for the file is resolved to a symbolic value, the engine is unable to execute that file. Because of these limitations, the symbolic-execution engine is unsound and incomplete. Nonetheless, our design strategies allow the engine to scale to real-world web applications.

CHAPTER 4. PARSING CONDITIONAL SYMBOLIC OUTPUT

To enable analysis on the generated client code, we first need to parse the output computed from the symbolic execution on the server-side code into a conditional DOM called *VarDOM* that compactly represents all variations of the generated client-side code. Let us first describe the VarDOM and subsequently explain how we derive it from the textual output of symbolic execution with *variability-aware parsing* [101].

4.1 VarDOM: Representation for Conditional DOM

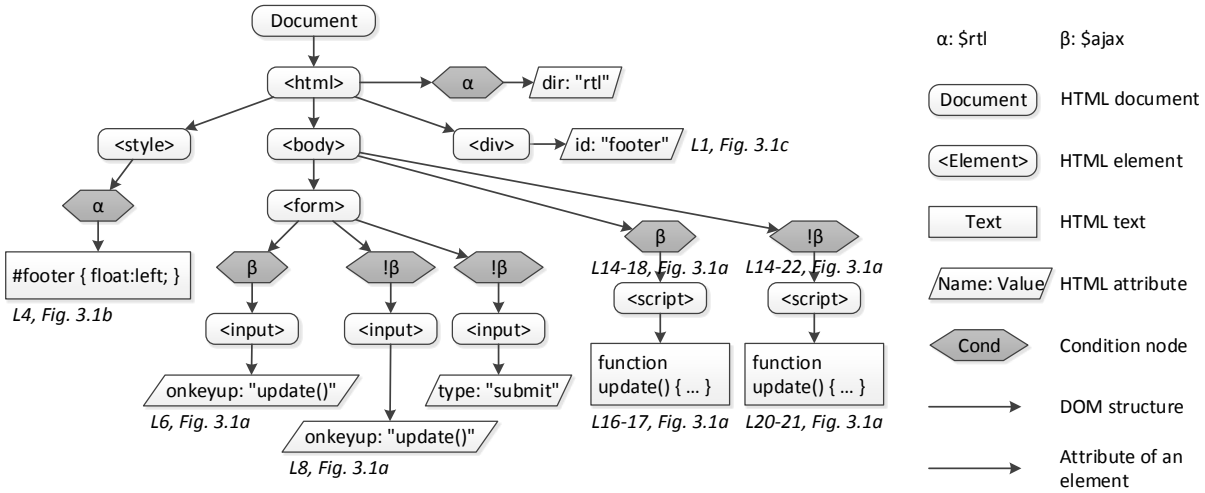


Figure 4.1 VarDOM: conditional representation for the client-side code of the PHP program in Figure 3.1

Just as the DOM represents the hierarchical syntactic structure of a web page with nested nodes of four types (HTML elements, attributes, text, and comments), a VarDOM

represents the tree structure of a web page with variations. The key difference is that in a VarDOM all elements can be *conditional*, that is, they are part of the web page only given a specific condition called *presence condition*. We model conditional elements with *condition nodes* in the tree, where the condition node holds the presence condition for its subtree, as illustrated in Figure 4.1. Note how this representation can compactly represent variations within similar pages; in our running example, possible client-side pages differ with regard to input fields depending on whether AJAX is enabled, but all pages share the same <html> and <body> elements. Conceptually, it is possible to move condition nodes up the VarDOM hierarchy by replicating code fragments yielding a less compact representation (see the choice calculus for a formal treatment [42]).

4.2 Variability-Aware Parsing to Construct VarDOM

To transform the character stream into a VarDOM that represents the structures of the client-side implementation, we use variability-aware parsing. Since the output of symbolic execution contains conditional characters (only printed under given path constraints) and symbolic characters, parsing is challenging. A regular parser can only be used on code without conditional text and therefore would not be able to parse such output. Fortunately, the problem resembles closely the challenge of parsing unpreprocessed C code that still contains #if directives, which has been solved recently [78]. To illustrate the similarities, we list the output of symbolic execution as code with #if directives in Figure 4.2.

In Figure 4.2, the path constraints are represented using #ifdef preprocessor directives. Next to a path constraint, we also show the source code of the corresponding PHP expression in a comment. (For readability, we omit origin information in this listing.) As can be seen in Figure 4.2, the texts on lines 3, 8, 14, 16, and 17 are optional depending on a condition, whereas the other parts are always present in the output. The

```
1  <html
2  #if α // $rtl
3  dir="rtl"
4  #endif
5  >
6      <style type="text/css">
7      #if α // $rtl
8          #footer {float : left ;}
9      #endif
10     </style>
11 <body>
12 <form method="Φ" name="searchform">
13     #if β // $ajax
14     <input ... onkeyup="update()"/>
15     #else
16     <input ... onkeyup="update()"/>
17     <input type="submit" />
18     #endif
19 </form>
20 <script type="text/javascript"> ...
```

Figure 4.2 An excerpt of the conditional output of the PHP program in Figure 3.1 represented with conditional-compilation directives (converted from the D–Model in Figure 3.2). Greek letters represent symbolic values.

output contains not only concrete strings but also symbolic values (e.g., Φ). Note that this (conditional, symbolic) output contains only textual information without syntactic information, i.e. all parts are treated as text regardless of whether they contain HTML, JS, or CSS code.

Next, we explain how we will use *TypeChef* to parse this text into a VarDOM with a lexer, a SAX parser, and a DOM parser.

1. Lexer. From symbolic execution's output (a D-Model), the lexer produces a sequence of conditional characters. The condition of each character is derived from the path constraint under which the output was produced during symbolic execution (represented with #if directives in Figure 4.2). We preserve the same formalism for formulas and satisfiability checking used during symbolic execution (propositional formulas and SAT solvers in our implementation). The lexer produces a special string value SYM for

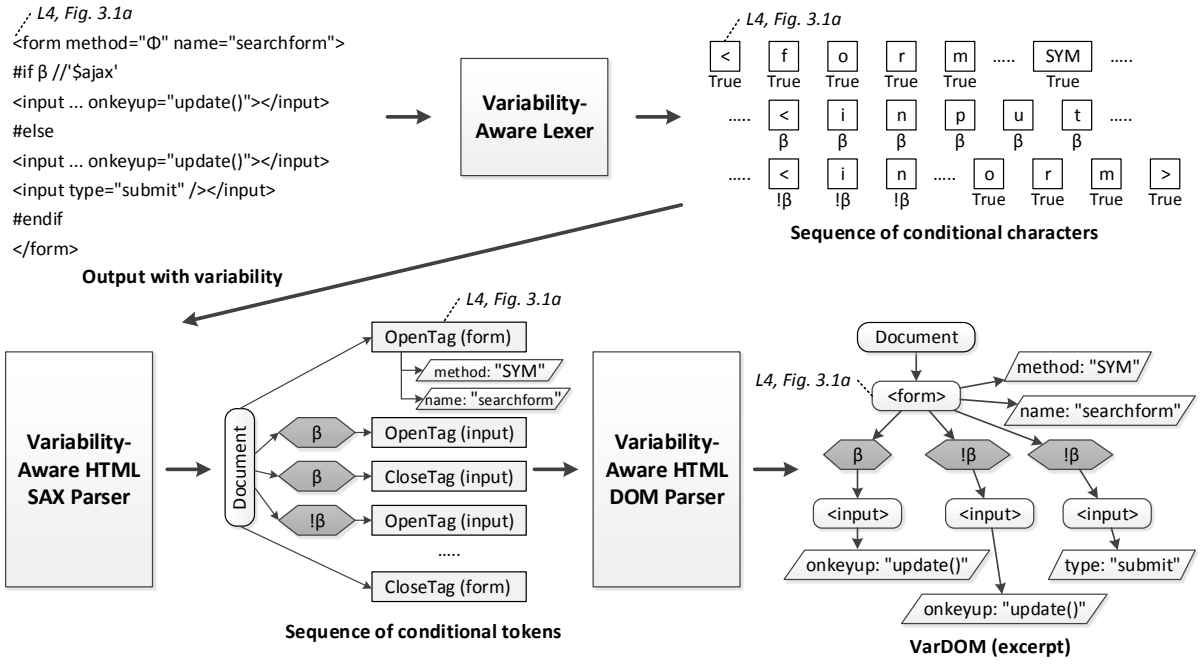


Figure 4.3 Parsing output with variability to build a VarDOM

symbolic values in the output and propagates origin locations from the PHP code for every individual character. We exemplify the step for our example in Figure 4.3.

2. SAX parser. To deal with the complexity of HTML, we proceed in two steps common in HTML parsers. The first step recognizes nodes and their attributes in a flat structure (SAX-style parsing) whereas the second step builds a tree from those nodes (DOM parser). The SAX parser takes the stream of conditional characters and produces a list of conditional nodes. Nodes can be opening tags with a name and possibly conditional attributes (e.g.,), closing tags with a name (e.g.,), or text fragments containing possibly conditional characters. The parser accepts symbolic tokens in text, as tag names, as attribute names, as attribute values, or as whitespace. As seen in Figure 4.3, this parsing step produces a very shallow parse tree of a document with a list of conditional tags, texts, and comments, of which start tags can contain conditional attributes, and texts/comments can contain conditional characters. The parser framework propagates origin locations and presence conditions.

3. DOM parser. In the next parsing step, we use the document's list of conditional nodes from the previous step as conditional token sequence for the subsequent DOM parser. The DOM parser itself is simple, since it recognizes only a tree structure based on matching starting and closing tags. However, the context-sensitive nature of checking well-formedness in HTML requires matching names in opening and closing tags, for which we wrote a simple combinator. Again, the parser framework propagates origin locations and presence conditions. In the VarDOM of our running example, exemplified in Figure 4.3, the first input field is guarded by condition node β ($ajax), and the other input fields are guarded by condition nodes $\neg\beta$ (!$ajax). (Note that we simply propagate attributes inside nodes and variations inside text nodes from the SAX parser.)

Reporting parsing errors. During parsing, the two parsers reject ill-formed HTML code (the first parser rejects invalid syntax of tags and attributes and the second parser rejects invalid nesting and missing closing tags). In each case, the parsers report a conditional error message for invalid configurations (with location information tracked from the initial PHP code) and a parse tree for the remaining configurations.For example, the HTML DOM parser would report a missing closing tag for in line 2 in the example below in the configurations with $C evaluating to true.

PHP code:		Output of symbolic execution:	
1	**if** ($C)	1	<form>
2	$div = ' ';	2	#**if** α // '$C'
3	**else**	3	
4	$div = '';	4	#**endif**
5	**echo** '<form>' . $div . '</form>';	5	</form>

Although relaxed or error-recovering parsing is conceptually possible, rejecting ill-formed code has the additional benefit of being able to report client-side errors during development while embedded in server-side code. In addition, we could easily check validity

and other invariants on top of the VarDOM representation, reporting conditional error messages when structural assumptions are violated.

Assumptions and limitations. The output of symbolic execution may contain symbolic values representing a potentially infinite number of possible client-side programs. The symbolic execution engine however explores only a finite number of paths in the server-side code (due to our simplifications with regard to loops and recursion, see Section 3), which allows us to parse the output into a structure with a finite number of variations expressed through condition nodes, assuming that symbolic values do not affect the output's structure. Specifically, we assume that symbolic values produce tag names, attribute names, or attribute values when used in that location and produce well-formed HTML fragments otherwise (in which case we can parse it as text or white space). For example, we fundamentally could not parse the output ' Ψ' if we had to assume that Ψ might provide the closing tag. Although it is easy to construct artificial examples that we cannot parse, we have not seen a code fragment in practice in which a symbolic value (e.g., from user input or another source of nondeterminism) affected the structural well-formedness of produced client-side code. In addition, to avoid the possible combinatorial explosion of alternatives, we enforce that HTML elements within the alternatives must have the same type (e.g., an HTML opening tag in the TRUE branch goes with an HTML opening tag in the FALSE branch).

CHAPTER 5. FOUNDATION FOR CROSS-LANGUAGE PROGRAM ANALYSIS TECHNIQUES

The VarDOM representation captures all DOM variants of the output of a server-side program and allows us to analyze the embedded code written in HTML/JS/CSS. In this section, we discuss novel concepts, techniques, and tools for cross-language program analysis in PHP web applications, including (1) building call graphs for embedded client code in different languages [101, 103], (2) computing cross-language program slices [102], and (3) computing a novel test coverage criterion called *output coverage* that aids testers in creating effective test suites for detecting output-related bugs.

5.1 Building Call Graphs for Embedded Client Code

The VarDOM is the basis for all subsequent analyses to build conditional call graphs that can be used for tool support. Even though not executable, we interpret relationships among HTML and CSS elements as call-graph edges as well, since they equally provide a foundation for corresponding editor services. In general, a conditional call graph consists of nodes reflecting positions in the server-side code. (Technically, a node can refer to multiple positions in the source code if it was concatenated from multiple string literals.) and edges representing relationships among those nodes (calls, corresponding tags, affected CSS rules, and so forth). Edges in a conditional call graph can have a presence condition, meaning that the node is related to another node only in certain executions of the server-side program. For example, an opening HTML tag can be closed by two

different closing tags resulting in two conditional call-graph edges, as illustrated with the script tag in our running example (see Figure 3.1). As usual, call graph edges can be navigated in both directions for different editor services, e.g., "jump to declaration" versus "find usage". If the server-side code has multiple entry points (e.g., multiple .php files a user can call), we build VarDOMs and analyze each entry separately and subsequently merge all call graphs, in which the nodes point to the same PHP code locations.

We illustrate three analyses to extract conditional call graphs for IDE support in HTML ("jump to opening/closing tag"), CSS ("jump to affected nodes" and "find applicable CSS rules"), and JS ("jump to declaration"), each while embedded in server-side code [101, 103].

5.1.1 Supporting HTML Jumps

We develop a call graph for HTML by defining *HTML jumps* from a source to a target, in which the source is an *HTML opening tag* and the target is its corresponding *closing tag* (either the entire tag or the name of the tag). This type of jump is useful in helping developers understand the structures of HTML tags that would be produced by their PHP program and find closing elements especially if they are generated from different PHP files, such as the body tag in our running example.

Building call-graph edges for HTML jumps is straightforward. We simply traverse the VarDOM and look up the origin locations of the opening and closing tags of each element (as explained in Section 4.2, tokens representing opening and closing tags produced by the SAX parser are used in the DOM parser). We create a call-graph edge between those locations with a presence condition reflecting the element's presence condition in the VarDOM (i.e., a conjunction of all condition nodes between the element and the VarDOM's root). When an element has alternative opening or closing tags, it occurs repeatedly in the VarDOM under alternative presence conditions (e.g., the two <script> elements in Figures 3.1 and 4.1).

5.1.2 Supporting CSS Jumps

CSS code consists of CSS rules to define styles for HTML elements. We create call-graph edges (or jumps) between CSS rules and all HTML elements selected by them, to support navigation among those elements within server-side code, similar to the debugging facilities that many browsers provide for generated client-side code.

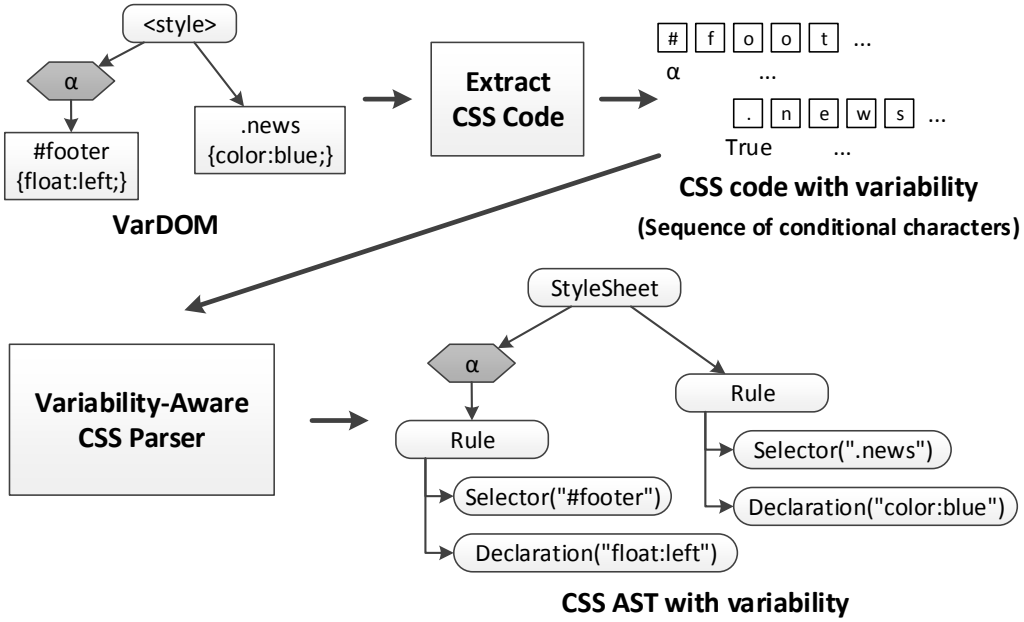


Figure 5.1 Parsing CSS code with variability

Unlike our HTML-jump analysis in which opening and closing tags are directly available in the VarDOM, we need to extract CSS rules from text fragments in the VarDOM—typically from <style> tags and from included (potentially generated) files. Notice that each CSS fragment is a sequence of conditional characters, in which we preserved the presence condition of the originating HTML element, and which may contain symbolic values. To analyze the CSS code with its variations, we wrote another variability-aware parser with TypeChef to recognize CSS as a list of conditional rules (ignoring symbolic values as white space), illustrated in Figure 5.1. The parser framework propagates origin locations and presence conditions.

After parsing, we only need to match the selector of each CSS rule against the VarDOM nodes. We create a call-graph edge for every match in the VarDOM, each with a presence condition that conjuncts the presence condition of the CSS rule with the presence condition of the matched HTML element. Edges with infeasible presence conditions can be filtered as far as the used formalism supports it. We reimplemented the matching algorithm for the most common selectors (class selectors, id selectors, element selectors, and nested selectors), implementing remaining selectors is technically straightforward following the specifications of CSS [141]. We list the CSS-related call-graph edge for our running example in Figure 3.1.

5.1.3 Conditional JS Call Graph

To build a conditional call graph for JS, in general, we can develop a variability-aware analysis from scratch in line with our solution for CSS. However, due to the complexity of building call graphs already for non-embedded JS code without conditional parts [44], we want to reuse existing infrastructures for building JS call graph. Our key idea is to reencode JS code with generation-time variability as JS code with runtime variability, in which the presence conditions of JS code are encoded as the conditions of regular JS `if` statements enclosing those code elements. After transforming the code into regular JS code (still tracking origin locations), we reuse WALA, an existing state-of-the-art tool for building a JS call graph [142]. Through consistent origin tracking, we can translate the identified call-graph nodes and edges back to their location in the VarDOM and hence also back to the original string literals in PHP code. We derive presence conditions for call-graph edges from the presence conditions of the involved VarDOM nodes, again filtering infeasible edges. Our goal is to provide editor support for JS code embedded in server-side code for every call-graph edge that WALA finds on the generated JS code.

1. Parsing JS code. As for CSS, we first extract all JS code fragments from text elements the VarDOM. Specifically, we collect the content of HTML <script> tags,

linked JS files, and all event handlers, such as onload and onclick. Again, we build a variability-aware JS parser on top of the TypeChef framework that accepts a sequence of conditional characters and produces a JS parse tree with conditional nodes (ignoring symbolic values). We follow the JS grammar specification [39], but ignore the context-sensitive semicolon-inserting feature in our prototype. To simplify subsequent steps, we push up conditional nodes to the level of statements, that is, variations inside statements are expanded to two alternative statements.

2. Reencoding variability. After parsing, we reencode generation-time variations (condition nodes with presence conditions in the AST) as runtime variations with if statements. This strategy, named *configuration lifting* or *variability encoding*, has been used in model checking and deductive verification of product lines to reuse analysis techniques that are oblivious to generation-time variations but can handle runtime variations [116, 136, 13, 12]. We reencode variability with the following two key transformation rules:

R1: Condition($cond$, Statement($stmt$)) $\rightarrow$

IfStatement(String($cond$), Statement($stmt$))

R2: Condition($cond$, FunctionDeclaration($name, params, body$))

$\rightarrow$ IfStatement(String($cond$), ExpressionStatement(

AssignmentExpression(Identifier($name$), "=",

FunctionExpression($params, body$))))

Rule R1 reencodes a statement $stmt$ under presence condition $cond$ (if $cond \neq$ true) as a JS if statement with a condition representing $cond$ and $stmt$ in the then branch (e.g., lines 12–14 of Figure 5.2). Rule R2 for a function declaration similarly reencodes function declaration as an equivalent assignment of a function expression inside an if statement (e.g., lines 13 and 18 of Figure 5.2).

Conceptually, it is possible to prove that a reencoding maintains the execution semantics of all configurations, e.g., by showing that the execution semantics of executing

```
1  <html dir=``rtl">
2      <style type=``text/css">
3          #footer {float: left}
4      </style>
5  <body>
6      <form method="Φ" name="searchform">
7          <input ... onkeyup="if ('β') update()"/>
8          <input ... onkeyup="if ('!β') update()"/>
9          <input type=``submit" />
10     </form>
11     <script type=``text/javascript">
12         if ('β') {
13             update = function () {...}
14         }
15     </script>
16     <script>
17         if ('!β') {
18             update = function () {...}
19         }
20     </script>
21     <div id=``footer"> ... </div>
22 </body>
23 </html>
```

Figure 5.2 Reencoding variability for JS code

the program is not affected by the reencoding in any configuration (where a configuration would either remove unnecessary code at generation time or initialize the configuration parameters equivalently to be interpreted at runtime). For our purpose, a strict notion of correctness is not necessary, since we perform an unsound call-graph analysis subsequently. It is sufficient to encode the program in a way that the analysis tool can identify the correct call-graph edges; hence we checked correctness of our encoding through testing only.

3. Reencoding HTML code. JS code can interact with the HTML DOM during execution (check existence of an element, access its properties, and so forth), which the used analysis framework WALA takes into account to some degree. WALA takes as input an HTML document and turns it into pure JS code which generates the document

using JS instruction document.createElement. WALA's call graph building algorithm for JS is then applied on this transformed JS code. Therefore, we reencode the entire HTML code by removing all condition nodes from the VarDOM. The resulting HTML document contains all possible alternative nodes in which all JS code is properly reencoded. This reencoding of HTML is a crude approximation, but sufficient for WALA's analysis in our experience. For instance, we reencode our running example as in Figure 5.2 and analyze it with WALA.

4. Call-graph generation with WALA. To create the actual call graph, we run WALA for JS [142] on the reencoded HTML/JS code. We take WALA's result as is and track nodes back to their origin in the VarDOM and in the PHP code. We create presence conditions for call-graph edges from the conjunction of the presence condition of the two involved VarDOM nodes. In our running example, WALA in fact ignores the conditions of the created if statements and creates a total of four call-graph edges from each update call to each function declaration. However, since both update calls and declarations depend on the same symbolic path condition in our example (i.e., we know that expression $ajax has the same value β in both server-side if statements), two call-graph edges receive infeasible presence condition $\beta \wedge \neg\beta$ and can be discarded. We show the JS call graph for our running example in Figure 3.1.

5.2 Cross-language Program Slicing

5.2.1 Concepts

In the literature, a *(forward) program slice* consists of the parts of a program that *may be affected* by the values computed at a *slicing criterion*, which is a point of interest typically specified by a program point and a set of variables [138]. Various program slicing methods have been proposed [138], since different properties of slices might be required for different applications. In this paper, we chose a class of program slicing

that is based on data dependencies. This class is called *thin slicing* [132] as opposed to traditional slicing based on both data and control dependencies, which typically produces slices that are too large to be useful for human inspection. A full slice can always be easily expanded from a thin slice [132]; we discuss this expansion in Section 5.2.3.5. Specifically, we define a (forward, thin) program slice with respect to a slicing criterion C (specified by the code location of a data entity) as *a set of definitions and references* of data entities in the web application that have *direct or indirect data-flow relations* from the value computed at C.

Table 5.1 Extension of data-flow relations for dynamic web applications

Relation	Within one language	Across languages
Def-use (Def. to ref. flow)	**F1.** A definition d and a reference r of a variable v have a *def-use* relation if there exists a control flow from the statement containing d to the statement containing r without intervening redefinitions of v. Alternatively stated, the definition of v at statement S_d is a reaching definition for S_r.	**F3.** A reference r and a definition d have a *cross-language def-use* relation if r and d are written in different languages, and r refers to the entity defined at d. For instance, the PHP variable `$_POST['update']` refers to the value of an HTML input named 'update'.
Info-flow (Ref. to def. flow)	**F2.** For a statement S, a reference r of variable v_1 has an *information-flow* relation with a definition d of variable v_2 if the value of v_1 on entry to S may affect the value of v_2 on exit from S. As an example, in the statement `$x = $y + $z`, the references `$y` and `$z` have information-flow relations with the definition `$x`.	**F4.** A reference r in language L_1 and a definition d in a different language L_2 have a *cross-language information-flow relation* if r generates r^*, and r^* forms the code that is used in the computation for the value of d. For example, in the PHP code `echo "<input name='input1' value ='$x'>"`, the value of the PHP variable `$x` is assigned to the value of the HTML input 'input1'.

Types of data-flow relations. We propose a program-slicing technique for dynamic web applications that is based on the relations between the definitions and references of data entities, namely definition-use (*def-use*) relations and information-flow

(*info-flow*) relations [21], which are traditionally used for analyzing programs written in a single language. In the context of dynamic web applications, we extend these relations also for entities that are written in different languages (see Table 5.1).

5.2.2 Approach Overview

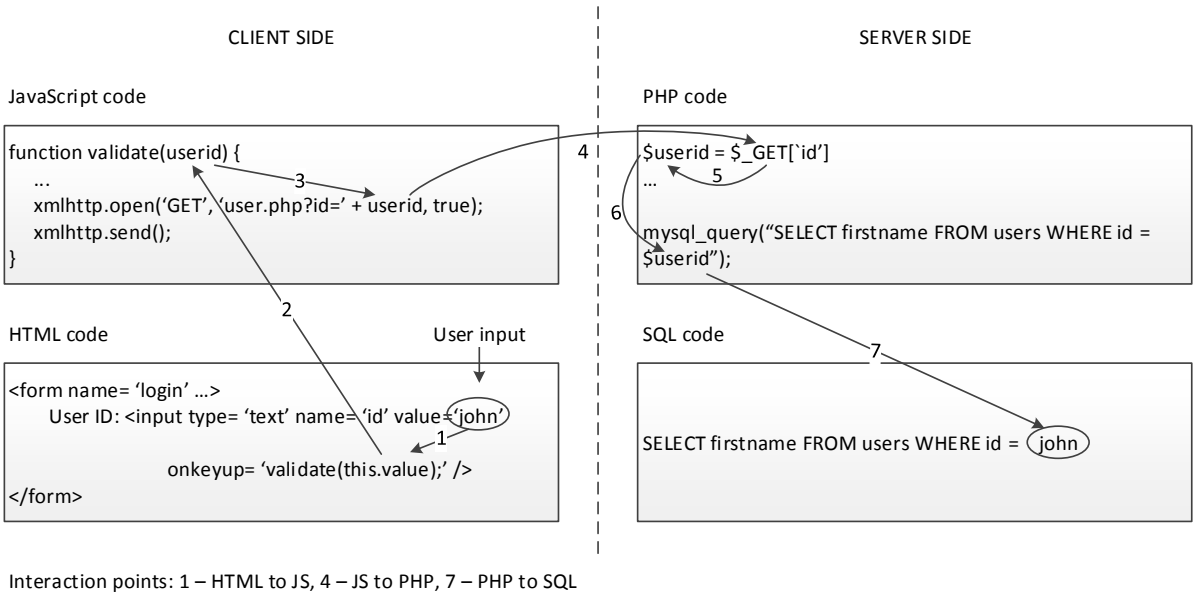


Figure 5.3 Example of a cross-language program slice

As explained in the introduction, program slicing for dynamic web code faces several challenges: (1) Data needs to be transferred back and forth between the server and client sides; control and data flows across different languages need to be taken into account when computing a program slice. (2) Client-side program entities (e.g., HTML input fields and JS variables) are often embedded in PHP string literals or computed via various string operations. (3) The control and data dependencies for the embedded code might be governed by the conditions in the server-side code (i.e., some of the dependencies are conditional). For illustration, Figure 5.3 shows an example of a cross-language program slice from the value 'john' of an HTML <input> element.

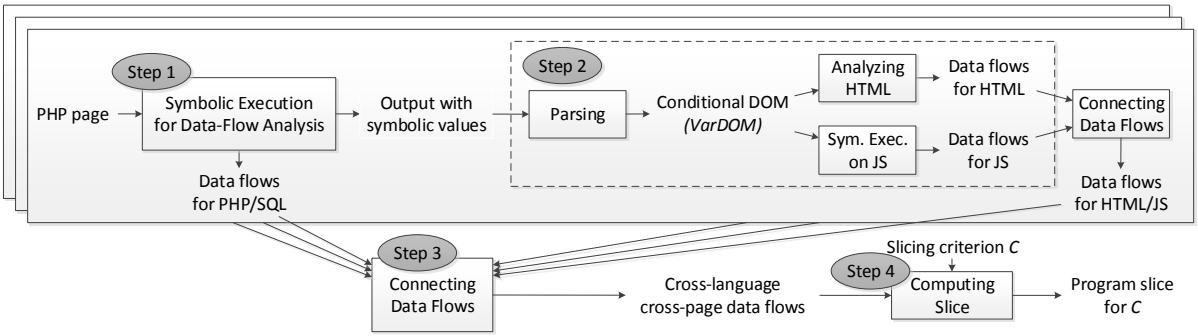


Figure 5.4 Overview of WebSlice

We propose WebSlice [102], an approach to compute program slices in a PHP web application. WebSlice proceeds in four main steps: (1) performing symbolic execution on the PHP code to approximate its output as well as constructing the data flows for server-side code in PHP and SQL, (2) parsing and analyzing the output to construct the data flows for client-side code in HTML and JS, (3) connecting the data flows across different languages, and (4) computing a slice given a slicing criterion. Figure 5.4 gives an overview of these steps.

Step 1—Symbolic execution for data-flow analysis. The goal is two-fold: (1) to approximate the output of a PHP program so that the data flows within embedded client code can be analyzed in later steps and (2) to construct the data flows within the server-side code. For approximating the output, we reuse our symbolic-execution engine [105]. Symbolic execution explores different paths in a PHP program and computes/propagates the values of definitions and references of data entities. Conveniently, this process allows us to track the data flows within the server-side code. Since we need our symbolic execution engine anyway to approximate the output, we reuse and extend it with new mechanisms to record the data flows within PHP as well as SQL code, which is embedded in PHP strings and is also resolved by symbolic execution. In addition, an advantage of using symbolic execution is that we can eliminate some infeasible flows by checking the satisfiability of the path constraints under which the data entities appear.

```
1 if ($_GET['user'] == 'admin')
2     $message = 'Welcome admin!';
3 else
4     $message = 'Access denied.';
5 if ($_GET['user'] == 'admin')
6         echo '<div class="msg-admin">' . $message . '</div>';
```

For example, in the code above, the PHP variable definition $message on line 4 does not have a def-use relation with the reference on line 6 since they are under different path constraints. Also, symbolic execution allows us to resolve dynamically included PHP files, thereby detecting data flows that would otherwise be missed. For scalability, we have made several approximations to our symbolic executor such as running at most two iterations of a loop and skipping recursive function calls; we discussed them at the end of Section 3.

Step 2—Embedded code analysis. Using the VarDOM representation of the client code, we are able to analyze the embedded code written in HTML/JS and build their respective data flows. Since HTML is a declarative language, we collect the definitions of HTML entities. For JS code, we compute its data flows using a light-weight symbolic-execution engine with an algorithm to build data flows similar to that for PHP.

Step 3—Connecting data flows. Data flows can exist among data entities of different languages and across different pages. Thus, we connect the data flows among those entities based on cross-language def-use and information-flow relations (**F3** and **F4** in Table 5.1). For instance, the input fields in the HTML form <form action = 'EditAnnouncements.php' ... > have cross-language def-use relations with the corresponding PHP $_GET/$_POST variables on the page 'EditAnnouncements.php' since the inputs field are submitted to that page. Through this step, we obtain the data flows for the entire web application.

Step 4—Computing slice. Once the data-flow graph is produced, we can use it to quickly compute *any* program slice. Given a definition or a reference C, the slice for C consists of the definitions and references that are reachable from C in the graph.

5.2.3 Data-Flow Analysis via Symbolic Execution

This section presents our algorithm to construct the data flows and to compute the output and SQL queries of PHP code. The algorithm is built on top of our symbolic-execution engine (see Section 3.2). To describe our technique, in addition to the existing notation, we introduce L as the set of all definitions and references. To detect data flows, the program state $(\mathcal{V}, \mathcal{D}, \pi)$ now includes a **definition store** $\mathcal{D} : N \mapsto \mathcal{P}(L \times \Pi)$ that maps each variable name to its set of definitions together with a path constraint under which each definition appears.

In Figure 5.5, we formalize the key evaluation rules for our symbolic-execution engine and highlight the parts that we extend to identify data-flow relations. The functions *addEntity* and *addRelation* are used to create the nodes and edges of the data-flow graph (the graph is a global data structure and is not shown in the program state).

5.2.3.1 Intraprocedural Data Flows (Rules 1–3)

During symbolic execution, we detect data flows by identifying def-use and information-flow relations among data entities (**F1** and **F2** in Table 5.1). For ***def-use relations***, since a reference could have multiple definitions (e.g., a PHP variable can be defined in different branches and then later accessed after the branches), we need to keep track of the *set* of definitions of each reference. Therefore, we maintain these sets via the definition store $\mathcal{D}$. When a reference r with name n is found under a path constraint π_r, we look up its definitions in the set $\mathcal{D}(n)$ and match π_r with the constraints of those definitions to retain only feasible relations. Specifically, a definition d with constraint π_d in $\mathcal{D}(n)$ has a feasible def-use relation with r if $\pi_d \wedge \pi_r$ is satisfiable. (In other words,

Initialization:

$$\mathcal{V}(x) = \perp \quad \boxed{\mathcal{D}(x) = \emptyset} \quad \pi = \mathsf{TRUE}$$

1. Variable Access:

$$\frac{\boxed{r = addEntity(\$n)} \quad \boxed{addRelation(d, r),\ \forall (d, \pi_d) \in \mathcal{D}(n), isSat(\pi_d \wedge \pi)}}{\langle \$n, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle \rightarrow \langle \mathcal{V}(n), \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle}$$

2. Assignment:

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \rangle \\ \boxed{d = addEntity(\$n)} \quad \boxed{addRelation(r, d),\ \forall r \in vars(e)} \quad \boxed{\mathcal{D}'' = \mathcal{D}'[n \mapsto \{(d, \pi)\}]}\end{array}}{\langle \$n = e, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle \rightarrow \langle v, \mathcal{V}'[n \mapsto v], \boxed{\mathcal{D}''}, \pi \rangle}$$

3. If Statement:

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad isSat(\pi \wedge \pi') \quad isSat(\pi \wedge \neg\pi') \\ \langle\!\langle s_1, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \wedge \pi' \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi \wedge \pi' \rangle\!\rangle \quad \langle\!\langle s_2, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \wedge \neg\pi' \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \boxed{\mathcal{D}_2}, \pi \wedge \neg\pi' \rangle\!\rangle \\ \mathcal{V}_3(x) = select(\pi', \mathcal{V}_1(x), \mathcal{V}_2(x)) \\ \boxed{\mathcal{D}_3(x) = \{(d, \pi_d \wedge \pi') | (d, \pi_d) \in \mathcal{D}_1(x)\} \bigcup \{(d, \pi_d \wedge \neg\pi') | (d, \pi_d) \in \mathcal{D}_2(x)\}}\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle\!\rangle \rightarrow \langle\!\langle select(\pi', c_1, c_2), \mathcal{V}_3, \boxed{\mathcal{D}_3}, \pi \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad \neg isSat(\pi \wedge \neg\pi') \\ \langle\!\langle s_1, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi \rangle\!\rangle\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad \neg isSat(\pi \wedge \pi') \\ \langle\!\langle s_2, \mathcal{V}', \boxed{\mathcal{D}'}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \boxed{\mathcal{D}_2}, \pi \rangle\!\rangle\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \boxed{\mathcal{D}_2}, \pi \rangle\!\rangle}$$

4. Function Declaration:

$$\frac{\lambda \text{ is a pointer to function } n(\$n_1, ..., \$n_m)\{s\}}{\langle\!\langle \mathsf{function}\ n(\$n_1, ..., \$n_m)\{s\}, \mathcal{V}, \boxed{\mathcal{D}}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathsf{OK}, \mathcal{V}[n \mapsto \lambda], \boxed{\mathcal{D}}, \pi \rangle\!\rangle}$$

5. Function Invocation:

$$\frac{\begin{array}{c}\lambda = \mathcal{V}_0(n) \quad \lambda \text{ is a pointer to function } n(\$n_1, ..., \$n_m)\{s\} \\ \langle e_i, \mathcal{V}_{i-1}, \boxed{\mathcal{D}_{i-1}}, \pi \rangle \rightarrow \langle v_i, \mathcal{V}_i, \boxed{\mathcal{D}_i}, \pi \rangle,\ \forall i \in [1..m] \quad \mathcal{V}_f(x) = \begin{cases} v_i & \text{if } x = n_i \\ \perp & \text{otherwise} \end{cases} \\ \boxed{d_i = addEntity(\$n_i),\ \forall i \in [1..m]} \quad \boxed{\mathcal{D}_f(x) = \begin{cases} (d_i, \pi) & \text{if } x = n_i \\ \emptyset & \text{otherwise} \end{cases}} \\ \boxed{addRelation(r, d_i),\ \forall r \in vars(e_i)\ \forall i \in [1..m]} \\ \langle\!\langle s, \mathcal{V}_f, \boxed{\mathcal{D}_f}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}_{f'}, \boxed{\mathcal{D}_{f'}}, \pi \rangle\!\rangle \\ \boxed{RET_{ref} = addEntity(n)} \quad \boxed{addRelation(RET_{def}, RET_{ref}), \forall (RET_{def}, \pi_d) \in \mathcal{D}_{f'}(\text{`RET'})}\end{array}}{\langle n(e_1, ..., e_m), \mathcal{V}_0, \boxed{\mathcal{D}_0}, \pi \rangle \rightarrow \langle \mathcal{V}_{f'}(\text{`RET'}), \mathcal{V}_m, \boxed{\mathcal{D}_m}, \pi \rangle}$$

Figure 5.5 Symbolic execution's evaluation rules to detect data flows (extensions to PhpSync in Section 3.2 are highlighted in gray)

6. Return Statement:

$$\frac{\langle e, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle \rightarrow \langle v, \mathcal{V}', \boxed{\mathcal{D}'}, \pi\rangle \quad \boxed{RET_{def} = addEntity(e)} \quad \boxed{addRelation(r, RET_{def}),\ \forall r \in vars(e)} \quad \boxed{\mathcal{D}'' = \mathcal{D}'[\text{'RET'} \mapsto \{(RET_{def}, \pi)\}]}}{\langle\!\langle \mathsf{return}\ e, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle\!\rangle \rightarrow \langle\!\langle \mathsf{RETURN}, \mathcal{V}'[\text{'RET'} \mapsto v], \boxed{\mathcal{D}''}, \pi\rangle\!\rangle}$$

7. Block of Statements:

$$\frac{\begin{array}{c}\langle\!\langle s_1, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi\rangle\!\rangle \quad \pi' = whenEqual(c_1, \mathsf{OK}) \\ isSat(\pi \wedge \pi') \quad \langle\!\langle s_2, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi \wedge \pi'\rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \boxed{\mathcal{D}_2}, \pi \wedge \pi'\rangle\!\rangle \\ \mathcal{V}_3(x) = select(\pi', \mathcal{V}_2(x), \mathcal{V}_1(x)) \\ \boxed{\mathcal{D}_3(x) = \{(d, \pi_d \wedge \pi') | (d, \pi_d) \in \mathcal{D}_2(x)\} \bigcup \{(d, \pi_d \wedge \neg\pi') | (d, \pi_d) \in \mathcal{D}_1(x)\}}\end{array}}{\langle\!\langle s_1 s_2, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle\!\rangle \rightarrow \langle\!\langle select(\pi', c_2, c_1), \mathcal{V}_3, \boxed{\mathcal{D}_3}, \pi\rangle\!\rangle}$$

8. While Statement:

$$\frac{\langle\!\langle \mathsf{if}\ (e)\ \{s\ \boxed{\mathsf{if}\ (e)\ s}\ \}, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}', \boxed{\mathcal{D}'}, \pi\rangle\!\rangle}{\langle\!\langle \mathsf{while}\ (e)\ s, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}', \boxed{\mathcal{D}'}, \pi\rangle\!\rangle}$$

9–11. Rules 9–11 are similar to those in Figure 3.3.

12. mysql_query:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi\rangle \rightarrow \langle query, \mathcal{V}', \mathcal{D}', \pi\rangle}{\langle \mathsf{msql\_query}(e), \mathcal{V}, \mathcal{D}, \pi\rangle \rightarrow \langle parseAndFindSqlDefs(query), \mathcal{V}', \mathcal{D}', \pi\rangle}$$

13. mysql_fetch_array:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi\rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi\rangle}{\langle \mathsf{msql\_fetch\_array}(e), \mathcal{V}, \mathcal{D}, \pi\rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi\rangle}$$

14. Array Access of SQL Data:

$$\frac{\begin{array}{c}\langle e_1, \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle \rightarrow \langle v_1, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi\rangle \quad \langle e_2, \mathcal{V}_1, \boxed{\mathcal{D}_1}, \pi\rangle \rightarrow \langle v_2, \mathcal{V}_2, \boxed{\mathcal{D}_2}, \pi\rangle \\ \boxed{v_1 \text{ is a set of SQL definitions}} \quad \boxed{d \in v_1} \quad \boxed{d \text{ has name/index } v_2} \\ \boxed{r = addEntity(e_1[e_2])} \quad \boxed{addRelation(d, r)}\end{array}}{\langle e_1[e_2], \mathcal{V}, \boxed{\mathcal{D}}, \pi\rangle \rightarrow \langle symbolic(e_1[e_2]), \mathcal{V}_2, \boxed{\mathcal{D}_2}, \pi\rangle}$$

Notation and auxiliary functions (in addition to those in Figure 3.3):
- $vars(e)$ returns a set of references appearing in an expression e (except for arguments of user-defined function calls).
- $parseAndFindSqlDefs(query)$ parses an SQL query and returns a set of SQL definitions (SQL table columns) in the query.
- $addEntity(e)$ creates and returns a new definition/reference from e.
- $addRelation(l_1, l_2)$ records a def-use/info-flow between l_1 and l_2.

Figure 5.5 (Continued)

Symbolic execution's evaluation rules to detect data flows (extensions to PhpSync in Section 3.2 are highlighted in gray)

there exists at least one execution path where both d and r appear), as shown in rule 1 of Figure 5.5).

To identify ***information-flow relations***, at a variable assignment, we record the information flow from the variables on the right-hand side to the one defined on the left-hand side (rule 2). Note that if the right-hand side of an assignment contains a user-defined function call, the arguments in the function call do not have *direct* information-flow relations with the defined variable; we detect their relations through *interprocedural* data flows instead (Section 5.2.3.2). We also update the definition store $\mathcal{D}$ with the new definition of the variable. If a variable is redefined through sequential statements, we overwrite its previous definitions with the new definition since values from the previous definitions can no longer be accessed. If a variable is defined/redefined in branches of a conditional statement, we keep the values/definitions of the variable independent in the branches but combine them after executing all branches. We describe the details next.

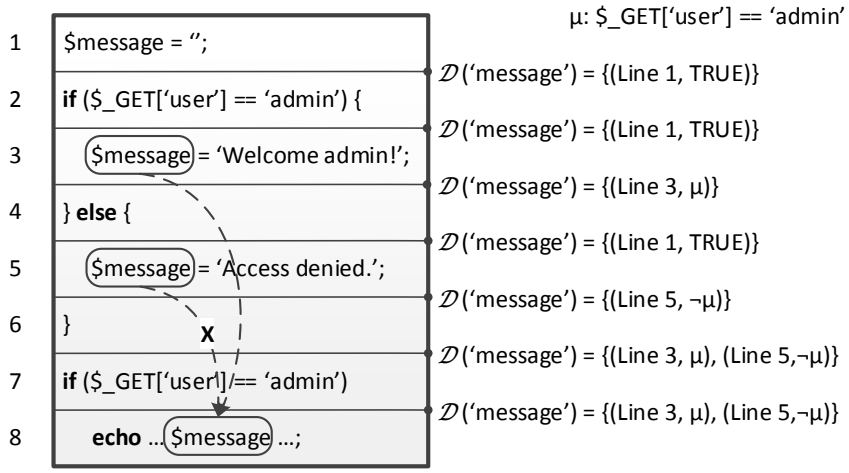


Figure 5.6 Detecting data flows at conditional statements

Handling conditional statements (rule 3). If the path constraints of both branches of an if statement are satisfiable, we explore both branches. Similarly to the value store $\mathcal{V}$, modifications to the definition store $\mathcal{D}$ take effect in the corresponding branch only. After executing the branches, we update the definition store with the com-

bined definitions from the two branches together with their corresponding constraints. Note that if the path constraint of one of the branches is unsatisfiable, we execute the other (satisfiable) branch only. As an illustration, in Figure 5.6, the variable $message after line 6 has two definitions from both branches. When the variable is accessed under constraint μ on line 8, we compare its constraint with the constraints of the definitions in $\mathcal{D}$ to eliminate an infeasible relation with the definition on line 5.

5.2.3.2 Interprocedural Data Flows (Rules 4–6)

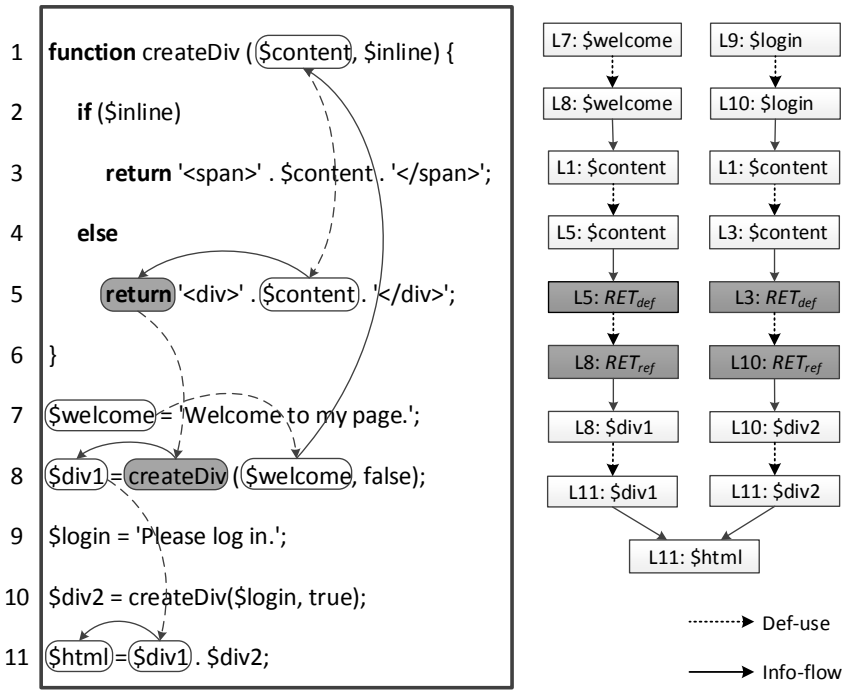


Figure 5.7 Interprocedural flows (RET nodes are highlighted)

Our extended algorithm instruments the function invocation process and tracks the data flows. We create a definition for each formal parameter and record the data flows from the arguments of the function call to the parameter definitions in the function declaration. To track the data flows from the function to its call site, we create two special RET nodes: a RET_{def} node representing the return value computed inside the function and a RET_{ref} node representing the propagated return value at the call site.

Note that if a function is invoked multiple times, we create separate entities, *RET* nodes, and data flows corresponding to each function invocation (for each invocation, the execution path in the function body could be different depending on the specific input arguments). Since we create different contexts at function calls, the approach does not suffer from the *calling-context problem* [138], caused by analyzing different function calls in the same context, which would result in infeasible interprocedural data flows. To illustrate, Figure 5.7 (right-hand side) shows the interprocedual data flows for the PHP variable $welcome (line 7) and $login (line 9). In the code, we show the data flow for $welcome only; the data flow for $login is similar. Note that one code location may correspond to several nodes in different contexts (e.g., the two nodes labeled L1: $content) since the createDiv function is executed twice. The details are shown in rules 4–6.

5.2.3.3 Handling Special Statements (Rules 7–8)

Handling a block of statements (rule 7). After executing the block, in addition to the value store, we also update the definition store in a similar manner.

Handling loops (rule 8). For a loop, our goal is to detect data flows across different iterations. For instance, in the code snippet below, there is a def-use relation from the variable $y on line 3 to the variable $y on line 2 if the loop can be executed multiple times.

```
1 while ($row = mysql_fetch_array($result)) {
2       $x = $y + 1;
3       $y = $x * 2;
4 }
```

Therefore, to detect such data flows, we execute the body of a loop *at most twice* by modeling the loop as two nested if statements and applying the rule for if.

Handling aliasing and objects (rule not shown). When a PHP object is created, we maintain two maps from the object's fields to their values and definitions (similar to the stores $\mathcal{V}$ and $\mathcal{D}$). Therefore, even if an object field is written and read via different variables (through aliasing), our algorithm can still recognize a def-use relation between the definition and reference, as illustrated below. (The same mechanism is used to handle assignment/call by reference.)

```
1 $x = new Foo(); $x->a = 1; $y = $x;
2 echo $y->a;
```

5.2.3.4 Data Flows between PHP and SQL (Rules 12–14)

In a web application, to retrieve data from a database, one can construct an SQL query and invoke PHP functions for database queries such as mysql_query. The returned data is stored in a *record set* with rows and columns. To iterate through each row in the record set, a PHP function such as mysql_fetch_array can be used. To access each column in a row, one can access the corresponding column name/index of the array containing the row. Since such an array access in PHP retrieves data originating from a database, we consider it as a data flow (def-use relation) from SQL to PHP. In that def-use relation, we consider the SQL table column name in the SQL query as an *SQL definition* and the corresponding array access as a *PHP reference to an SQL entity*.

To detect such data flows, during symbolic execution, we input the value of an SQL SELECT query, which could also contain symbolic/conditional characters, into our variability-aware SQL parser (similar to the HTML parser in Section 4.2) to recognize table column names as SQL definitions (function $parseAndFindSqlDefs$ in rule 12 of Figure 5.5). This set of SQL definitions is propagated through mysql_fetch_array function calls (rule 13). When there is an array access to such SQL data, we detect a relation between them (rule 14). In this work, we detect data flows from SQL to PHP; we

plan to apply similar ideas for data flows within SQL and from PHP to SQL (via SQL INSERT/UPDATE statements).

5.2.3.5 Traditional vs. Thin Slicing

We compute a thin slice by including all reachable nodes from a given node in the data-flow graph. However, we could easily record control dependencies for traditional slicing as follows. At an if statement (rule 3), we could additionally record the control dependencies between references on the if condition and the definitions within its branches and extend our graph to have both control and data dependencies on entities (similar to a PDG on statements). We can then reduce program slicing to a reachability problem on this graph.

5.2.4 Embedded Code Analysis

We parse the symbolic output of a PHP program with our HTML and JS variability-aware parsers [101] into a VarDOM representation of the client-side code (Figure 4.1). We then analyze the VarDOM to collect data entities and construct data flows for the embedded code.

Analyzing HTML. Since HTML is a declarative language, we detect the *definitions* of HTML entities by traversing the VarDOM tree and identifying the following types:

(1) HTML definitions by name: These entities are identified by the 'name' attribute of an HTML element (e.g., <form name='*form1*'>).

(2) HTML definitions by ID: These entities are identified by the 'id' attribute of an HTML element (e.g.,).

(3) HTML definitions by URL parameters: These entities are detected in HTML query strings (e.g., the data entity lang in <a href = 'index.php?*lang*=en'>).

Building data flows for JS. To construct data flows for JS, we first extract JS code from JS locations on the VarDOM. These locations include HTML <script> tags

and HTML event handlers (e.g., onload, onclick). The VarDOM already contains the parsed JS ASTs for these code fragments [101], each of which serves as an entry point. We then use a light-weight symbolic-execution engine for JS that is similar to the one for PHP (by adapting the rules in Figure 5.5 for JS), run it for every entry point and detect data flows. Currently, we do not handle client code that is dynamically generated from JS code such as document.write or eval, and data flows involving AJAX.

5.2.5 Cross-language Data Flows

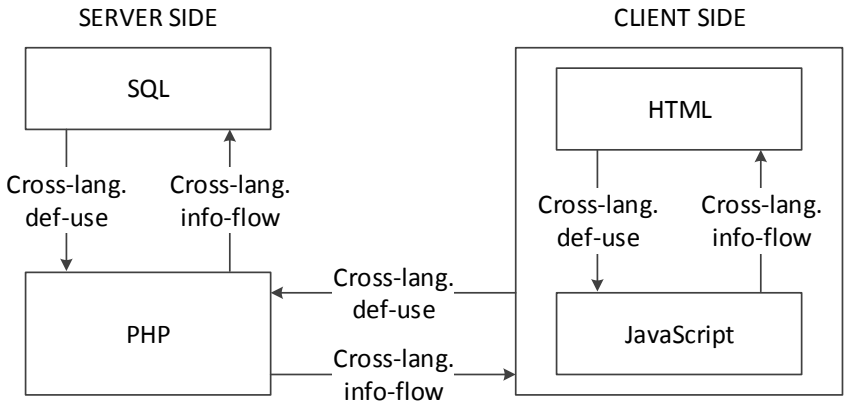


Figure 5.8 Data-flow relations across different languages

Data flows can exist among entities of different languages (**F3** and **F4** in Table 5.1). In Figure 5.8, we show all possible def-use and information-flow relations across languages. We detect those cross-language flows as follows.

F3—Cross-language def-use relations. Table 5.2 shows the types of cross-language def-use relations in a web application.

(1) Between HTML/JS and PHP (rows 1–3): A PHP program can access data sent from a client page via PHP $_POST/$_GET or $_REQUEST arrays (corresponding to HTTP POST/GET protocols or both). These arrays hold key/value pairs, where the keys are the names of the HTML input fields. Therefore, we identify those array accesses as *PHP references to client-side entities*. Note that the submitted destination of the

Table 5.2 Types of cross-language def-use relations

	Ref.	Definition	Reference example	Definition example
1	PHP	HTML input	**$_GET['input1']**	<input name='input1' **value**='0'...>
2	PHP	HTML URL	**$_GET['input1']**	<a href='index.php?**input1**=0'>
3	PHP	JS	**$_GET['input1']**	**document.form1.input1.value**='0'
4	PHP	SQL	**$row['column1']**	SELECT **column1** FROM table1
	JS	HTML by name:		
5	JS	- form	document.**form1**	<form name='**form1**'...>
6	JS	- input	doc...form1.**input1**	<input name='**input1**'...>
7	JS	- input value	doc...input1.**value**	<input name='input1' **value**='0'>
8	JS	HTML by ID	document.getEle...ById('**id1**')	

client-side entities (specified by the 'action' attribute of an HTML form or the address part in a URL) must match the PHP page containing the PHP reference.

(2) Between SQL and PHP (row 4): As described in Section 5.2.3.4, we detect these relations during our symbolic execution on PHP.

(3) Between HTML and JS (rows 5–8): In the client code, JS can operate on HTML elements via the HTML DOM. For example, the JS expression document.form1.input1.value retrieves the value of an HTML input field named 'input1' in a form named 'form1'. We identify these JS expressions as *JS references to HTML entities*. However, if they appear on the left-hand side of an assignment, we consider them as *JS definitions of HTML entities* instead since they redefine the values of the corresponding HTML entities. Similar to detecting data flows in PHP, we also check the path constraints under which these client-side entities are generated to eliminate infeasible data flows among them.

F4—Cross-language information-flow relation. During symbolic execution on PHP or JS, we track any generated string value (or symbolic value) to the variable or expression that generates it. If the value is used in an information-flow relation in the generated code, we recognize it as a cross-language information-flow relation from the generating language.

We apply the above process for a (predefined) set of page entries (PHP files that can be requested by a web browser) to build the data flows within individual pages (the data flows for a page can involve multiple files). To detect data flows across page entries, we detect types 1–3 in Table 5.2. (The other types in Table 5.2 are applicable for within-page relations only.) Data flows via cookies and sessions are currently not supported. Note that the resulting data-flow graph may contain *identical clusters* of nodes where there are no edges across those clusters and the clusters all correspond to the same code locations in the server-side program (since the same code might be executed multiple times); in such cases, we retain only one cluster and discard the others.

Calling-context problem with inter-page data flows. When a client page submits data to the server side, the corresponding server-side program is invoked to handle the request. Conceptually, this process is similar to invoking a function call from the client page in which the arguments to the function call are the client's data. Although we could handle the invocation of pages similarly to function calls, in our current implementation, we do not execute a page entry multiple times. Thus, the calling-context problem may occur for inter-page data flows, resulting in some infeasible data flows. However, our test results on a real-world system indicated that this problem does not cause significant imprecision.

5.3 Output-Oriented Testing

5.3.1 Motivation

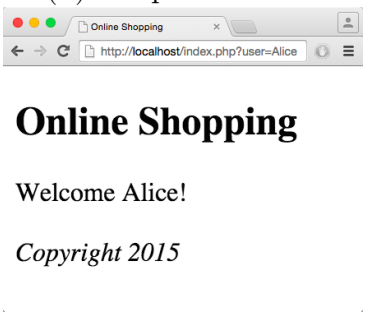
In software development, different testers have different focuses and priorities. Testing may focus on functional correctness, but also on performance, security, usability, accessibility, layout quality, and other quality attributes. Whereas code-level testing is well supported by tools and coverage metrics, we argue that testers inspecting the *output* of a program are much less supported, even though the output is the main prod-

uct of many software systems, including dynamic web applications. For example, for a UI tester tasked with checking a web application for layout issues or proof reading all texts, we cannot offer 'coverage' measures that would be available for a traditional software tester. For such testers, code measures such as line or branch coverage represent a wrong abstraction, refer to artifacts not directly related to the testing tasks and possibly unknown to the tester, and incentivize testing priorities inefficiently as we will show. Instead, we propose coverage measures for the program's output, to support, what we call, *output-oriented testing*.

(a) Example PHP program

```
1 echo "<h2>Online Shopping</h2>";
2 if (isAdmin()) logAdminAccess();
3 if (isLoggedIn())
4     echo "<div>Welcome " . $user . "!</div>";
5 else
6     echo "<div>Please log in first.</div>";
7 echo "<em>Copyright 2015</em>";
```

(b) Output of a test



(c) Output coverage visualization

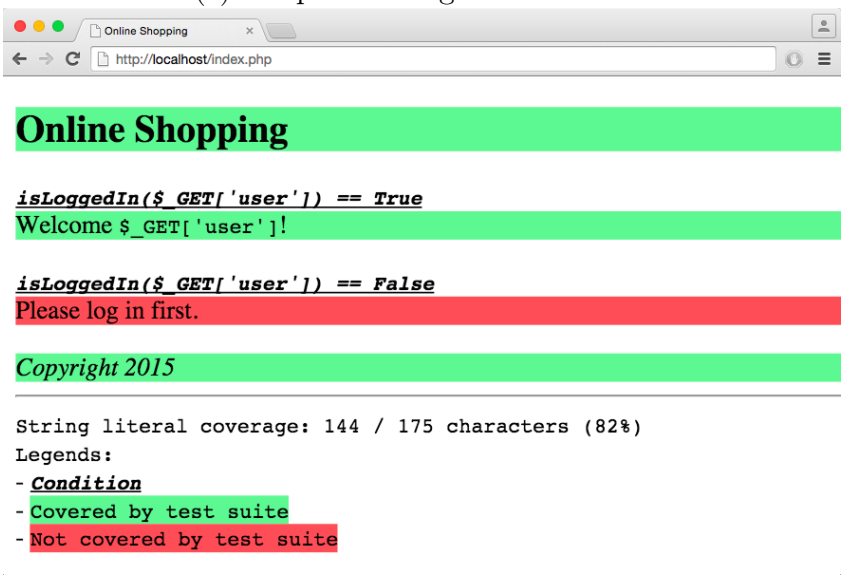


Figure 5.9 An example web application and illustration of output coverage

For testers that focus primarily on the output of an application (UI testers and many others), we propose coverage metrics on the output of an application, which indicate how much of the possible output has been produced by a set of tests and what output is still uncovered. *Output coverage metrics* measure coverage in terms of the output, not in terms of the code that produces the output, because code that does not produce any output is irrelevant to a UI tester. For example, to inspect all possible outputs of our example program in Figure 5.9a for layout issues, a UI tester would not care about the implementation logic beyond understanding which different pages can be produced and would not care about whether admin access is logged (line 2). The UI tester would care about seeing all text *in context* to *investigate different parts of the output and the consistency among those parts*, for example, ensuring that font sizes are consistent in all outputs. With an *output coverage metric*, a tester can identify missing parts of the output and can prioritize tests to cover the largest parts of the output (not most statements of the implementation) first.

5.3.2 Output Coverage Metrics

5.3.2.1 Representing the Output Universe

A key step in determining output coverage is to compute the *output universe*, that is, the set of all possible outputs from a web application. Unlike traditional code coverage metrics, such as statement coverage or branch coverage, where the set of all statements and branches in a program is well defined and finite, the set of all possible outputs is usually infinite due to unknown data coming from user inputs and databases. Such unknown data can lead to different outputs with the same structure (e.g., only the user's name changes while the layout of the web page and the welcome message remain the same), or different outputs with different structures (e.g., a different set of functionality is presented depending on whether the user has the administrator role). To address this challenge, we reuse our existing tree-based approximation of all possible outputs, called

D-Model (see Section 3.1). The key idea in our approximation is that we abstract unknown data as symbolic values. When string values are produced under specific path constraints (depending on one or more symbolic values), we keep all alternatives and their associated constraints. In this way, the D-Model approximates all possible structures of the output with all statically determinable content, leaving symbolic values as 'placeholders' for unresolved data.

To illustrate our representation of all possible outputs, let us consider a slightly extended version of our initial example. As shown in Figure 5.10a, we include a second decision about displaying coupons. In Figure 5.10b, we show our corresponding output universe representation with Greek letters for symbolic values and #if directives for alternatives depending on a path constraint. Note that we maintain for each character in the output universe its origin location in the server-side program obtained via our symbolic execution (not shown).

5.3.2.2 Family of Output Coverage Metrics

Just as a family of coverage metrics such as statement, branch, and path coverage has been defined on code [97, 10], we propose *a family of coverage metrics that are applied to the output* of a web program. We define output coverage for a given test suite as follows:

Coverage of string literals. Coverage for string literals (Cov_{str}) is similar to statement coverage in that we measure how much of the content contained in string literals in the program (that are relevant for the output universe) have been produced in at least one test case. The rationale for this type of coverage is that a string that *could appear* in the output *needs to be tested* in a concrete output generated by at least one test case. Note that not all string literals in the PHP program contribute to the output: For example, the literal "user" in our example is an array access but not part of the output universe and as such not measured by this metric. To measure the coverage for strings, we compute their covered length, or equivalently, the number of covered characters:

(a) A PHP program (extended from Figure 5.9) to illustrate different output coverage metrics

```
1  echo "<h2>Online Shopping</h2>";
2  if (isAdmin($_GET["user"]))
3      logAdminAccess();
4  if (isLoggedIn($_GET["user"]))
5      $message = "<div>Welcome " . $_GET["user"] . "!</div>";
6  else
7      $message = "<div>Please log in first.</div>";
8  echo $message;
9  if ($showCoupons)
10     echo "<div>Coupons are available!</div>";
11 echo "<div><em>Copyright 2015</em></div>";
```

(b) Representation of the output universe with CPP #if directives and symbolic values in Greek letters; origin locations of strings literals in server code are shown in italics

```
1  <h2>Online Shopping</h2> L1
2  #if α // isLoggedIn($_GET['user']) L4
3      <div>Welcome β!</div> // β represents $_GET['user'] L5
4  #else
5      <div>Please log in first.</div> L7
6  #endif
7  #if γ // $showCoupons L9
8      <div>Coupons are available!</div> L10
9  #endif
10 <div><em>Copyright 2015</em></div> L11
```

(c) A concrete output of a test case in which the user (Alice) is logged in as administrator and the $showCoupons option is enabled

```
1  <h2>Online Shopping</h2>
2  <div>Welcome Alice!</div>
3  <div>Coupons are available!</div>
4  <div><em>Copyright 2015</em></div>
```

(d) The S-Model for a concrete output

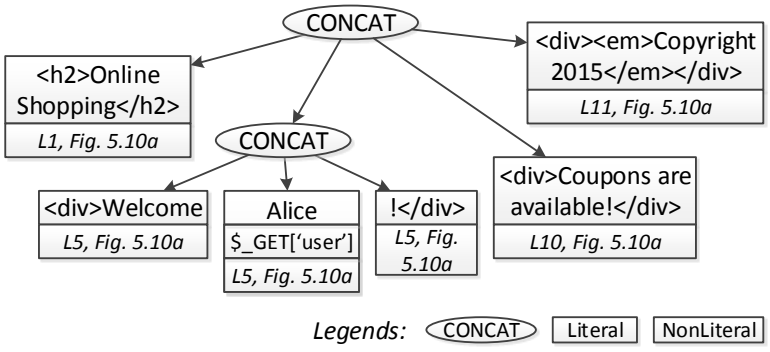


Figure 5.10 An example PHP program, its output universe representation, and S-Model

Definition 2 (Cov_{str}) *The ratio between the number of characters in string literals in the output universe that are covered by the test suite and the total number of all characters in the output universe.*

We define Cov_{str} based on the lengths of the literals, rather than the number of literals because a web application often has long literals containing large portions of HTML/JS code. Counting characters aligns better with the chance of bugs since long literals are more likely to contain bugs than short ones.

Compared to a simpler approach of investigating all string literals in a program individually, testing with output coverage ensures that each literal is produced in the context of at least one full page. Such context allows to investigate presentation issues depending on context as font sizes, colors, surrounding texts, or ordering that cross multiple string literals. For example, a tester may want to assure that all tags are correctly nested, which is difficult to assess from looking only at individual string literals. Analogous to statement coverage, to achieve Cov_{str} coverage each literal has to appear only in a single context, not in all possible contexts.

Coverage of output decisions. In addition to covering all string literals, testers might also want to investigate the composition of the page when certain parts are not displayed based on some decision. For instance, they might want to check that the layout of the web page is still correct without the coupon output. We consider every control-flow decision in the program that affects the output as an *output decision*. A control-flow decision that does not produce output in either branch is not considered as an output decision. We define coverage on output decisions (Cov_{dec}) analogous to branch coverage on code:

Definition 3 (Cov_{dec}) *The ratio between the number of output decisions covered by the test suite and the total number of all output decisions in the output universe.*

Coverage of contexts. The previous measures consider only some contexts in which string literals may appear (i.e., at least one context for each string literal or output decision), while certain bugs in the output may appear in some contexts and not others. (Each context is a specific *combination of output decisions*, which can be listed by traversing the output decisions on the output universe.) Therefore, we also consider all possible contexts and define a corresponding output-coverage metric Cov_{ctx} analogous to path coverage on code:

Definition 4 (Cov_{ctx}) *The ratio between the number of combinations of output decisions covered by the test suite and the total number of all possible combinations of output decisions in the output universe.*

Table 5.3 Output coverage and code coverage for the example in Figure 5.10

Output coverage		**Code coverage**	
Cov_{str}	111 / 142 (78.1%)	Statement cov.	9 / 10 (90%)
Cov_{dec}	2 / 4 (50%)	Branch cov.	3 / 6 (50%)
Cov_{ctx}	1 / 4 (25%)	Path cov.	1 / 8 (12.5%)

In our example, there are three control-flow decisions (*if* statements) of which two affect the output; these two output decisions can be combined to provide four different contexts (in contrast to eight paths). As with path coverage, the number of contexts grows very quickly, such that achieving full Cov_{ctx} coverage is typically not a realistic goal. In Table 5.3, we show output coverage metrics and their code coverage counterparts for a single test case on our example.

Relations with code coverage metrics. Our three output coverage metrics are inspired by but do not necessarily correlate with statement coverage, branch coverage, and path coverage, respectively. Specifically, not all statements and decisions in a web application contribute to the output (e.g., the logging functionality in our example), and single statement can produce output for multiple string literals (e.g, the `echo` statement

on line 8 of Figure 5.10a generates the two alternatives on lines 2–6 in Figure 5.10b). Our output coverage metrics provide a tailored view on the test suite of a web application for testers who are focused on the output.

5.3.3 Computing Output Coverage

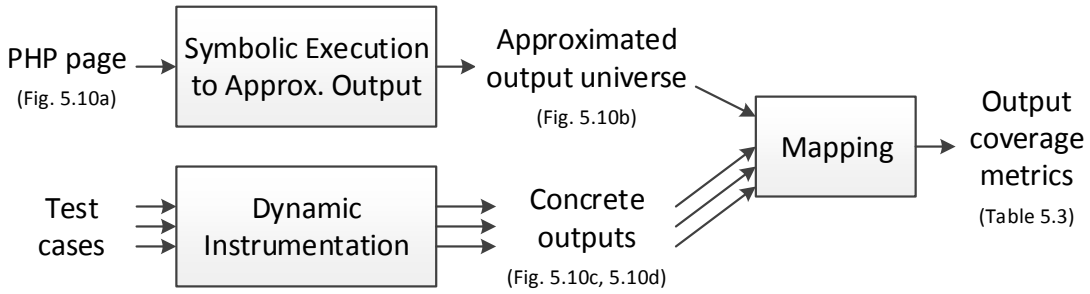


Figure 5.11 Computing output coverage

Key Idea. We compute output coverage metrics in three steps illustrated in Figure 5.11. First, we use our symbolic execution engine for PHP [105] to approximate the output universe of a PHP web application. Throughout symbolic execution, we track origin information for every character in the output. Second, we instrument a regular PHP interpreter to record the execution of a test case and the generation of its output. We again track origin information about every character in string values (either to string literals or to outside sources as user input). Third, we match concrete outputs from test executions on the output universe to measure coverage. The location information produced as part of every test execution is used to avoid ambiguity in identifying parts in the output universe that have been covered by the test suite. Since the number of string literals, output decisions, and contexts in the output universe representation is finite, we can compute the three coverage metrics as percentages.

Information about our symbolic execution technique has been explained in Section 3; next, we explain steps 2 and 3 in more detail.

5.3.3.1 Dynamic Instrumentation of Test Cases

To enable more precise mapping of test outputs on the output universe, we track *origin information* of string values that are output by concrete test executions. (Origin information is used for distinguishing string literals that have the same value but are produced from different locations in the server-side program.) String values in PHP are either introduced in literals or read from variables or functions that represent environment values, database results, or user input (e.g., `$_GET['user']`). Once created, we track origin information also through assignments, concatenation, and function calls until it is finally used as part of an `echo` or `print` statement (or inline HTML) to produce output.

We track origin information by attaching location information to string values. Technically, we attach a tree-based representation for string values called *S-Model* (short for 'string model'). An S-Model may contain three kinds of nodes: `Literal` and `NonLiteral` nodes with origin information and `Concat` containing nested S-Model nodes. In Figure 5.10d, we illustrate the S-Model corresponding to the output of Figure 5.10c of our running example (for clarity, we show only line-level location information).

We compute origin information with an instrumented PHP interpreter, based on the open-source interpreter Quercus [118]. We track the attached S-Model information during the evaluation of the test as shown in Figure 5.12. Specifically, we handle three kinds of expressions:

1. For a PHP literal expression (lines 2–6), we attach a new `Literal` node to the string value, pointing to the expression node with its location information.

2. For a PHP concatenation expression (lines 9-15), we create a `Concat` node to represent the returned string value with its children being the corresponding S-Models of the constituent sub-strings.

3. For all other PHP expressions, such as a PHP variables or a function calls (lines 18-22), we reuse the original interpreter's code to evaluate the expression and obtain

```
1  // Evaluating a Literal  Expression
2  function Value eval(LiteralExpr literalExpr)
3      value ← evalLiteral(literalExpr)
4      value.SModel ← new Literal(literalExpr)
5      return value
6  end
7
8  // Evaluating a Concat Expression
9  function Value eval(ConcatExpr concatExpr)
10     leftValue ← eval(concatExpr.LeftExpr)
11     rightValue ← eval(concatExpr.RightExpr)
12     value ← evalConcat(leftValue, rightValue)
13     value.SModel ← new Concat(leftValue.SModel, rightValue.SModel)
14     return value
15 end
16
17 // Evaluating a Non−Literal Expression
18 function Value eval(NonLiteralExpr nonLiteralExpr)
19     value ← evalNonLiteral(nonLiteralExpr)
20     if value.SModel is not set
21         value.SModel ← new NonLiteralNode(nonLiteralExpr)
22 end
```

Figure 5.12 Algorithm to create S-Model (added instrumentation is in italics)

its value (line 19). We attach a NonLiteral node referring to the corresponding non-literal expression that creates the value.

The output of the test is collected as a single large S-Model from echo and print statements, collecting all individual string outputs with Concat nodes.

5.3.3.2 Mapping Outputs to Output Universe

To compute output coverage, we identify which string literals/characters in the output have been covered by outputs generated in a test case. We map Literal nodes of the S-Model for test executions against the output universe to identify covered string literals and output decisions. An S-Model can be considered as a sequence of string literals (with location information), whereas the output universe can be considered as a sequence of string literals *and* symbolic values, with some parts being alternatives to one another.

In most cases, the mapping is straightforward because we simply can match string literals by location information. Matching is more difficult when concrete values of an execution are matched against symbolic values in the output universe, since there might be multiple possible matches. In addition, location information is not always available—there are cases where we might lose location information during processing when strings are processed through library functions. Therefore, for those cases where location information is not available or not sufficient, we use heuristic strategies to determine the best mapping among possible alternatives.

To illustrate our mapping challenge for cases involving symbolic values, consider mapping a string value "Welcome guest" with the following output universe. The string could be matched against either "Welcome β" assuming that α is TRUE or the literals "Welcome guest" if α is FALSE:

```
1 Welcome
2 #if α
3     β // $_GET['user']
4 #else
5     guest
6 #endif
```

To perform mapping in those cases, we use the following heuristic strategies:

- **Pivot mapping:** We first identify *pivots*—the string literals in the S-Model that can be mapped exactly to the string literals in the output universe. The remaining unmapped string literals will correspond to symbolic values or parts with alternatives. For instance, in the above example, we first map the string "Welcome" to the corresponding string on the output universe. The unmapped string "guest" will then be matched against the #if block in the output universe.

- **Best local mapping:** To identify which one of the alternatives should be mapped to a given string value, we recursively map the string value with each alternative and select the one with the best mapping result (the highest number of mapped characters). Note that we perform the mapping locally for (possibly nested) alternatives after pivots are identified in the previous step; we do not consider globally optimal mapping of alternatives. In this example, the string "guest" will be matched with both values in the true branch (β) and false branch ("guest") of the #if block for comparison.

- **Location mapping:** Since mapping can be ambiguous without location information (e.g., the string value "guest" can be mapped to either the symbolic value β or the string literal "guest"), we use the location information provided by the S-Model of a string value to compare with the location of a string value or symbolic value in the output universe to select a correct mapping. In this example, considering the location, the string "guest" can be mapped to only one of the two branches.

For our running example (with its output universe and S-Model shown previously in Figures 5.10b and 5.10d), our mapping algorithm proceeds as follows: First, the strings "<h2>Online Shopping</h2>" and " Copyright..." are mapped with the corresponding strings on the output universe. The remaining strings are first mapped to the first #if block. In the true branch, the strings " Welcome" and "! " are mapped with the corresponding strings while "Alice" is mapped to the symbolic value β (representing $_GET['user']). In the false branch, the strings cannot be mapped to " Please log in first ". Therefore, we select the mapping in the true branch. Finally, the string " Coupons are available! " is matched to the second #if block; it is then mapped to the string in the true branch (the false branch is empty).

Discussion. Since our output mapping algorithm works heuristically, it is important that there are sufficient pivots to guarantee that the local best mappings are correct.

As there are often large chunks of texts that remain unchanged across different executions, these pivots makes our mapping algorithm fast and highly precise. Note that our output mapping algorithm is extended from the CSMap algorithm described in our prior work [105]. While CSMap maps the output universe with a string in the output (without location information), our extended algorithm maps the output universe with an S-Model of a string value with its location information to avoid any ambiguous mappings.

CHAPTER 6. DEVELOPING DYNAMIC WEB DEVELOPMENT SUPPORT AND IDE SERVICES

Based on our program analysis infrastructure, we develop various types of support for dynamic web development activities such as IDE services, fault localization, bug detection, and testing.

6.1 IDE Services for Embedded Client Code

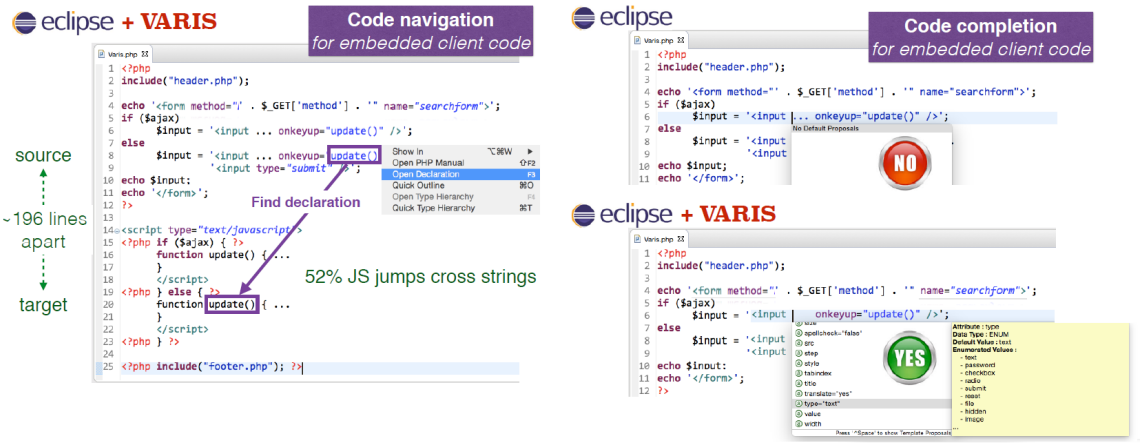


Figure 6.1 IDE services for embedded client code in dynamic web applications

In software development, IDE services such as syntax highlighting, code completion, and "jump to declaration" are used to assist developers in programming tasks. In dynamic web applications, however, since the client-side code is dynamically generated from the server-side code and is *embedded* in the server-side program as string literals, providing IDE services for such embedded code is challenging. In this work, we intro-

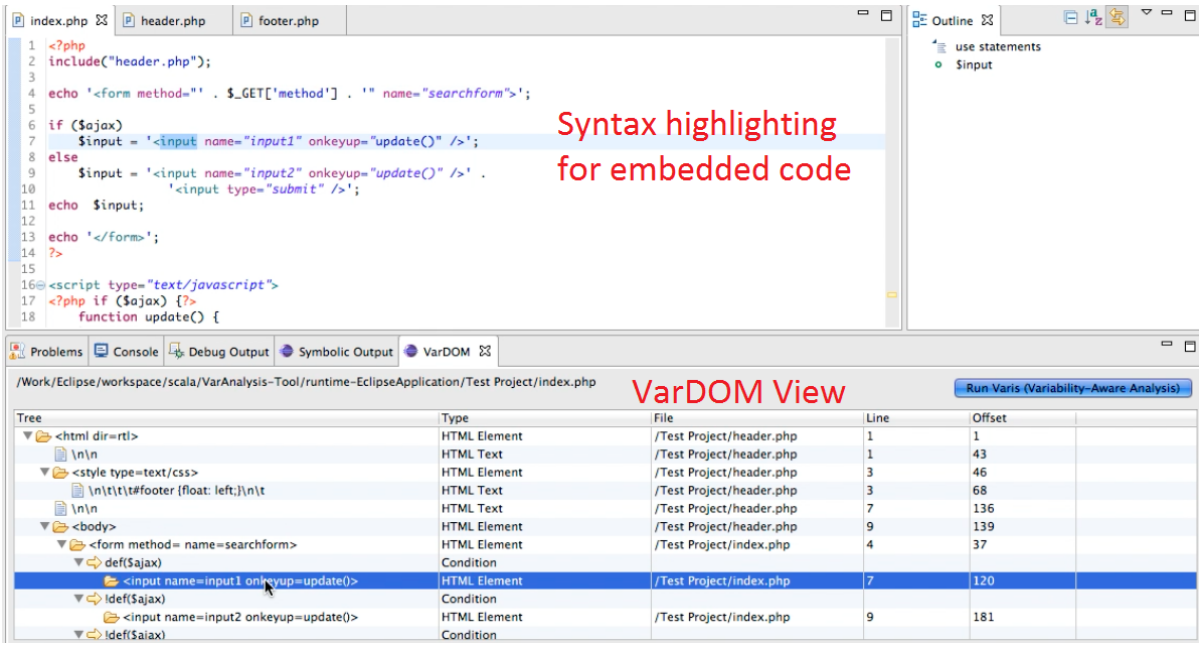


Figure 6.2 The VarDOM view and syntax highlighting support

duce *Varis* [101, 103] and *BabelRef* [106], the two tools that provide editor services on the client-side code of a PHP-based web application, while it is still embedded within server-side code. We implement various types of IDE services for embedded client code including syntax highlighting, code completion, "jump to declaration", and refactoring (see a screenshot in Figure 6.1).

6.1.1 The VarDOM View

When a PHP program is loaded and the Varis tool is enabled, Varis analyzes the PHP program and displays its VarDOM tree in an Eclipse tree view (the lower half in Figure 6.2). Each HTML element is displayed with its textual content, type, and location in the PHP code. Condition nodes in the VarDOM are annotated with an arrow. When the user selects an HTML element in the VarDOM view, its corresponding text in the source code will be highlighted. For instance, the highlighted HTML element in Figure 6.2 shows that it is located on line 7 of the PHP file and the label of its parent condition node shows that the element is generated when $ajax evaluates to TRUE.

Figure 6.3 Code completion support

6.1.2 Syntax Highlighting

With the type and location information of HTML elements in the VarDOM, Varis is able to highlight syntactic elements in the embedded client code. For instance, the HTML opening tag, the HTML attribute name, and HTML attribute values are colored differently on line 4 of Figure 6.2. Note that even though the HTML tag at line 4 is split into three fragments in the server code with some unknown data ($_GET['method']), Varis is still able to highlight the code elements correctly.

6.1.3 Code Completion

When the user points to a position inside a PHP string and invokes code completion support, Varis recognizes the code element of the embedded code at that location and provides a list of code recommendations for the code element as if it was written on static client code (without being embedded in a string). As an illustration, in Figure 6.3, since the user requests code completion support at a position after the HTML input tag name, Varis recommends attribute names that an HTML input can have. We use W3C standards on HTML elements and their attributes for embedded HTML code completion.

6.1.4 Jump to Declaration

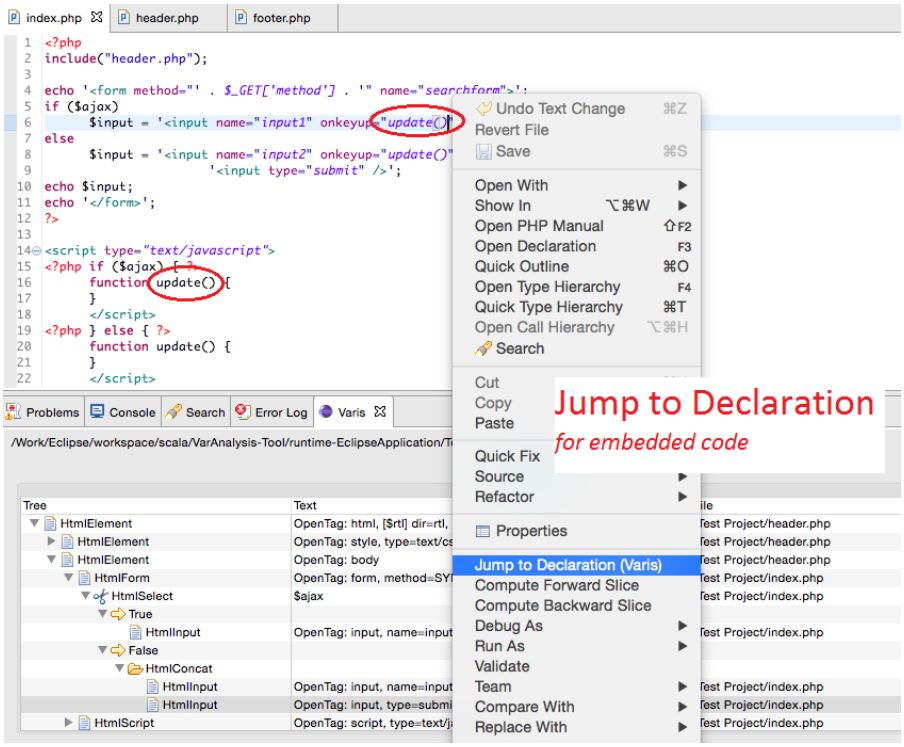


Figure 6.4 "Jump to declaration" support

Based on the underlying call graph created from the VarDOM, Varis allows a user to navigate between sources and targets in calling relationships: from JS function calls to their declarations, from opening to closing HTML tags, and from CSS rules to selected HTML elements. In Figure 6.4, when the user selects the JS function call update, a context menu appears allowing the user to use the "Jump to Declaration" functionality, which would take the user to the function declaration on line 18. Note that although another function declaration with the same name exists on line 22, Varis is able to pinpoint correctly the target by matching conditions between the function call site and its declarations. If one source has multiple targets, Varis will show the condition of the jump and allow a user to select the conditional navigation. For example, Varis will show the AJAX option and allow the user to choose the navigation from the opening <script> tag at line 14 of Figure 3.1 to its respective closing tag at either line 18 or 22.

6.1.5 Refactoring

The refactoring support, given by the BabelRef tool, provides two key features: detecting and displaying cross-language program entities/references and renaming those entities/references on request.

6.1.5.1 Entity Detection

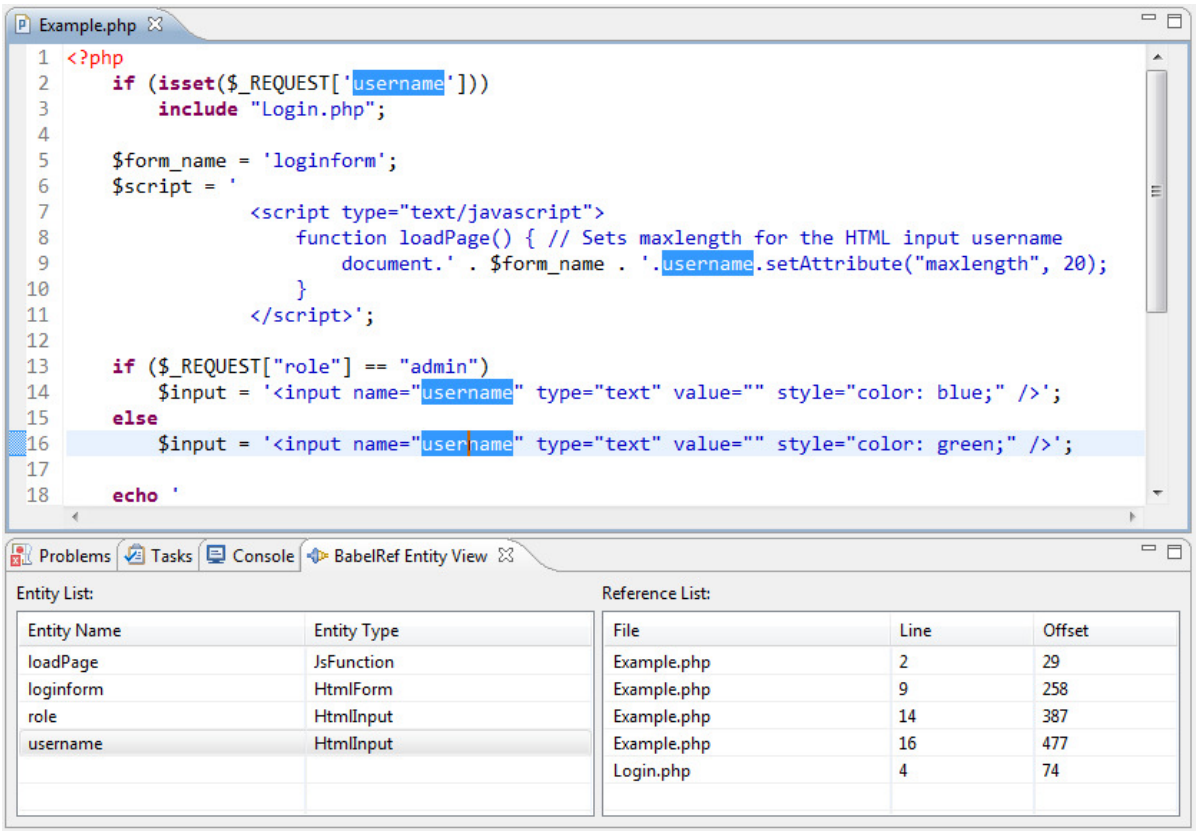


Figure 6.5 BabelRef's entity view

Figure 6.5 shows BabelRef's entity view with an entity list and a reference list. The entity list displays all the cross-language program entities in the currently-edited PHP file. When an entity is selected, the reference list gives the location information of all the references to that entity, including the source files, line numbers, and offset positions. For example, the PHP file in Figure 6.5 contains four entities, three of which are HTML

entities and the other is a JS function. The HTML entity username has five references located in two different PHP source files. As a user selects an entity in the editor window, all of its references in the file will be highlighted.

6.1.5.2 Entity Renaming

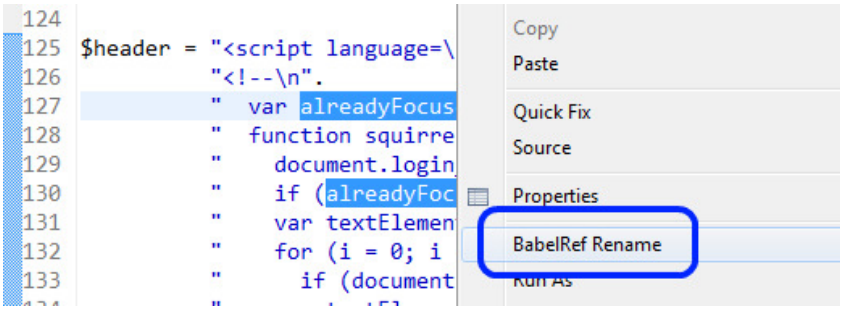


Figure 6.6 BabelRef's entity renaming: selecting an entity to rename

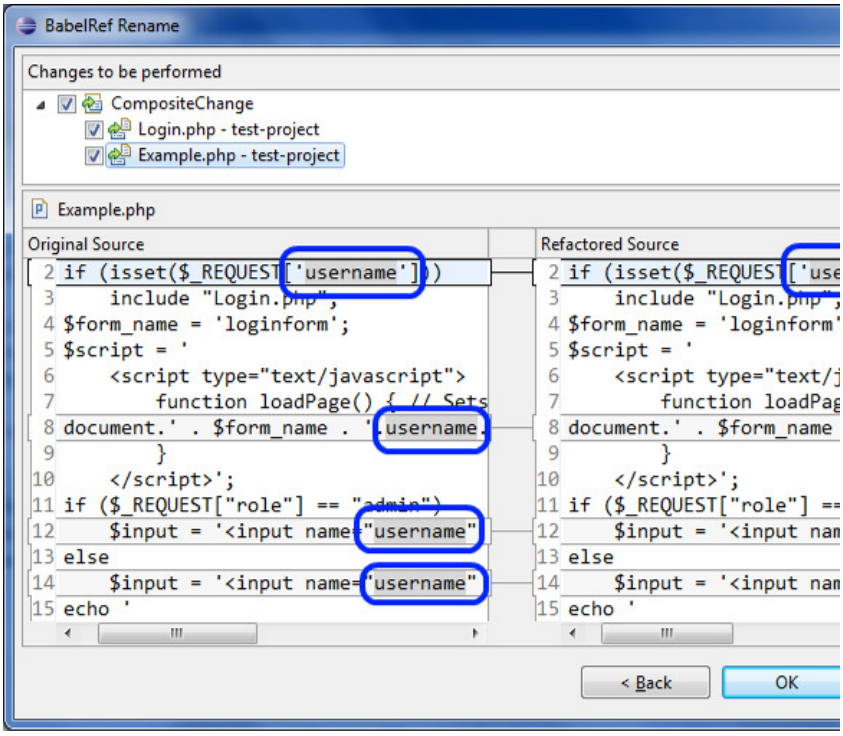


Figure 6.7 BabelRef's entity renaming: previewing changes

Based on the identified reference locations of the entities, BabelRef provides automatic renaming support on those entities. When the user right-clicks on an entity in the Eclipse editor, the BabelRef Rename command appears in the context menu allowing the user to rename the entity (Figure 6.6). The user can then enter a new name for the entity and preview the changes before applying the renaming operation (Figure 6.7).

After a renaming operation or whenever the user edits the source code, BabelRef automatically re-executes the PHP program symbolically, re-detects the entities in the background, and updates the entity and reference lists on the fly. In our experiments on several real-world web applications of size up to 50 KLOC, BabelRef normally took less than 40 seconds to perform symbolic execution and detect entities for a new system (with up to 300 entities and more than 2,000 references), and it took less than one second to re-detect entities and display the updated results when the source code changed.

6.1.5.3 Case Studies

a) References of alreadyFocused in JS code

```
$header = "<script language=\"JavaScript\" type=\"text/javasc
          "<!--\n".
          "  var alreadyFocused = false;\n".
          "  function squirrelmail_loginpage_onload() {\n".
          "    document.login_form.js_autodetect_results.valu
          "    if (alreadyFocused) return;\n".
          "    var textElements = 0;\n".
          "    for (i = 0; i < document.login_form.elements.l
          "      if (document.login_form.elements[i].type ==
          "        textElements++;\n".
```

b) References of alreadyFocused in HTML code

```
            html_tag( 'td',
addInput($user, $value, 0, 0, ' onfocus="alreadyFocuse
            'left', '', 'width="70%"' )
            ) . "\n" .
            html_tag( 'td',
addPwField($pw, null, ' onfocus="alreadyFocused=true;"
addHidden('js_autodetect_results', SMPREF_JS_OFF).
```

Figure 6.8 Cross-language entities/references in SquirrelMail-1.4.22

In SquirrelMail-1.4.22, alreadyFocused is a JS variable declared and used inside the JS code that is embedded in the PHP string $header (Figure 6.8a). The string value of $header will then be output to the client page. On the client page, there also exist two HTML elements produced by the PHP functions addInput and addPwField (Figure 6.8b). Using symbolic execution, BabelRef can construct the HTML code of these elements and detects that inside the event handlers for onfocus, the JS variable is accessed. Therefore, BabelRef recognizes all the four references of alreadyFocused.

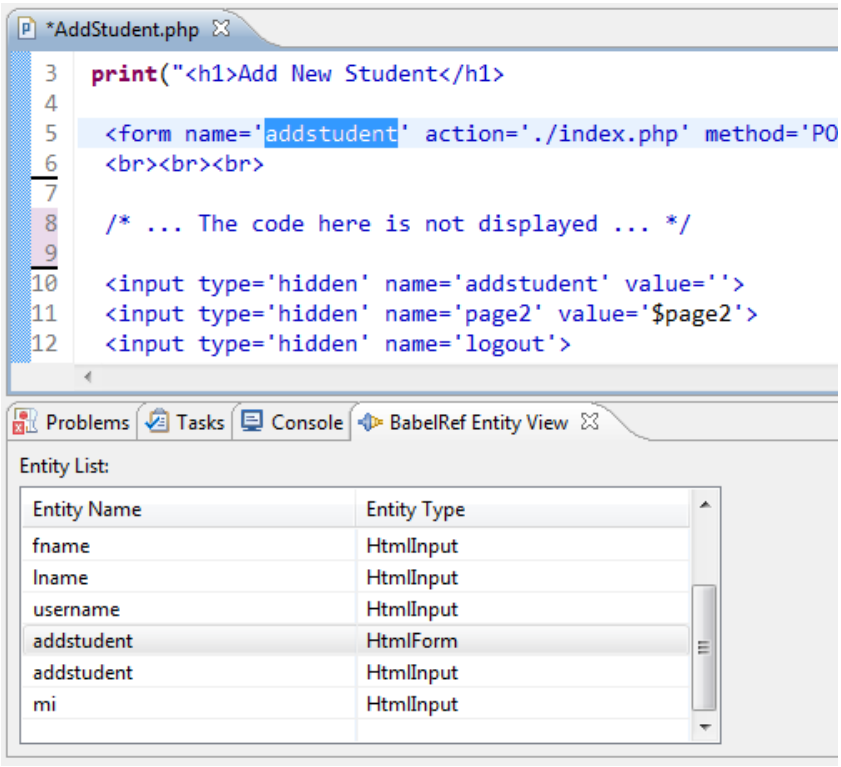


Figure 6.9 Entities with the same name in SchoolMate-1.5.4

In SchoolMate-1.5.4, there are two entities with the same name addstudent, one is an HTML form and the other is an HTML input field (Figure 6.9). If the user is interested in one of the entities only, BabelRef can help in identifying all the references belonging to the chosen entity. Thus, the user does not have to filter the results returned by a text search for the string "addstudent" to eliminate irrelevant references.

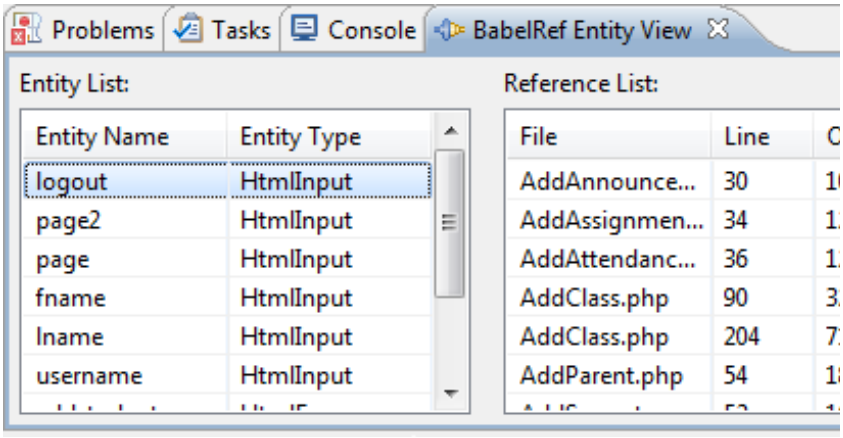


Figure 6.10 Entities with scattered references in SchoolMate-1.5.4

Also, entities may have references located at various locations. As can be seen in Figure 6.10, the references to the entity logout are scattered across more than 50 source files. If the user selects one of these references to rename, BabelRef will rename all the other references as well.

6.2 Fault Localization via Cross-language Program Slicing

We implemented our *WebSlice* approach as an Eclipse plug-in [102] (Figure 6.11). WebSlice extends our previous symbolic-execution engine (Section 3) and variability-aware parsers (Section 4). We use TypeChef's library for propositional formulas [78] with a JavaBDD backend [67] for tracking path constraints and checking satisfiability. Given a data entity requested by the user, WebSlice displays all definitions and references of the program slice for the entity in an Eclipse tree view, with parent-child relations showing data-flow relations between the entities. When the user selects a definition/reference in the slice, WebSlice highlights the corresponding element in the source code. In Figure 6.11, WebSlice displays the slice for the PHP variable $_POST["delete"], which has a direct information-flow relation with the variable $id and an indirect relation with the HTML input 'userid' embedded in PHP code (visible in the Eclipse view but not

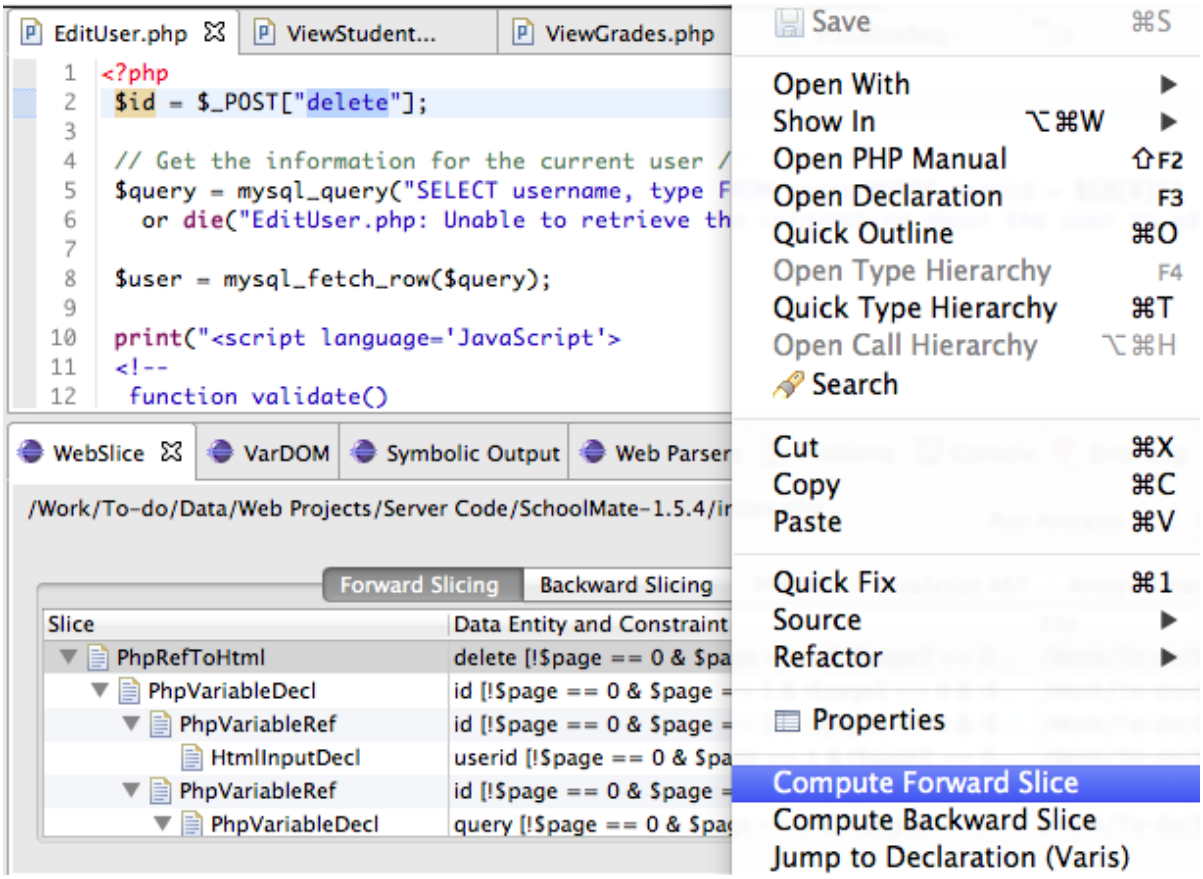


Figure 6.11 The WebSlice Eclipse plug-in

visible in the source code). The user can also request a *backward slice* for a data entity; in that case, WebSlice will show the slice in another tab. This type of cross-language slicing through embedded code was not possible with traditional editors. Such a slice produced by our tool can assist developers in debugging a program error manifesting at a given program entity. The developer can investigate the slice to identify potential faults resulting in the error.

6.3 Bug Detection

6.3.1 Dangling Reference Detection

We introduce *Dangling Reference Checker (DRC)* [107, 104], a tool to statically detect PHP and embedded dangling references in PHP-based web applications. DRC matches the constraint of each reference against those of their declarations to check if a reference is dangling. There are two cases in which a reference becomes dangling: (1) there exist no declarations with the same name and type as the reference; and (2) such declarations are found, but the combined constraint of all the matched declarations is stricter than the constraint of the reference (i.e., there exist some program executions in which the reference exists while its declarations do not).

We implemented DRC as a plug-in to the Eclipse environment. The key features in DRC include (1) displaying program entities/references and their constraints, and (2) detecting dangling references of both PHP and embedded types.

6.3.1.1 Entity Table

When a PHP program is loaded, DRC analyzes it and displays the list of entities (declarations and references) in an Entity Table (Figure 6.12). Each entity is displayed with its name, type, code location, and path constraint. For example, in Figure 6.12, the entity $input at line 33 of Login.php is a declaration of a PHP variable with its path constraint corresponding to the branching conditions at lines 26 and 32.

6.3.1.2 Dangling References

Using the Entity Table, DRC matches each reference with the corresponding declarations by names, types, and constraints. In the Dangling Reference View tab (Figure 6.13), the declarations and references of the same entity are grouped together. When the user selects an entity's name on the left-hand-side panel (Entity List), the list of its decla-

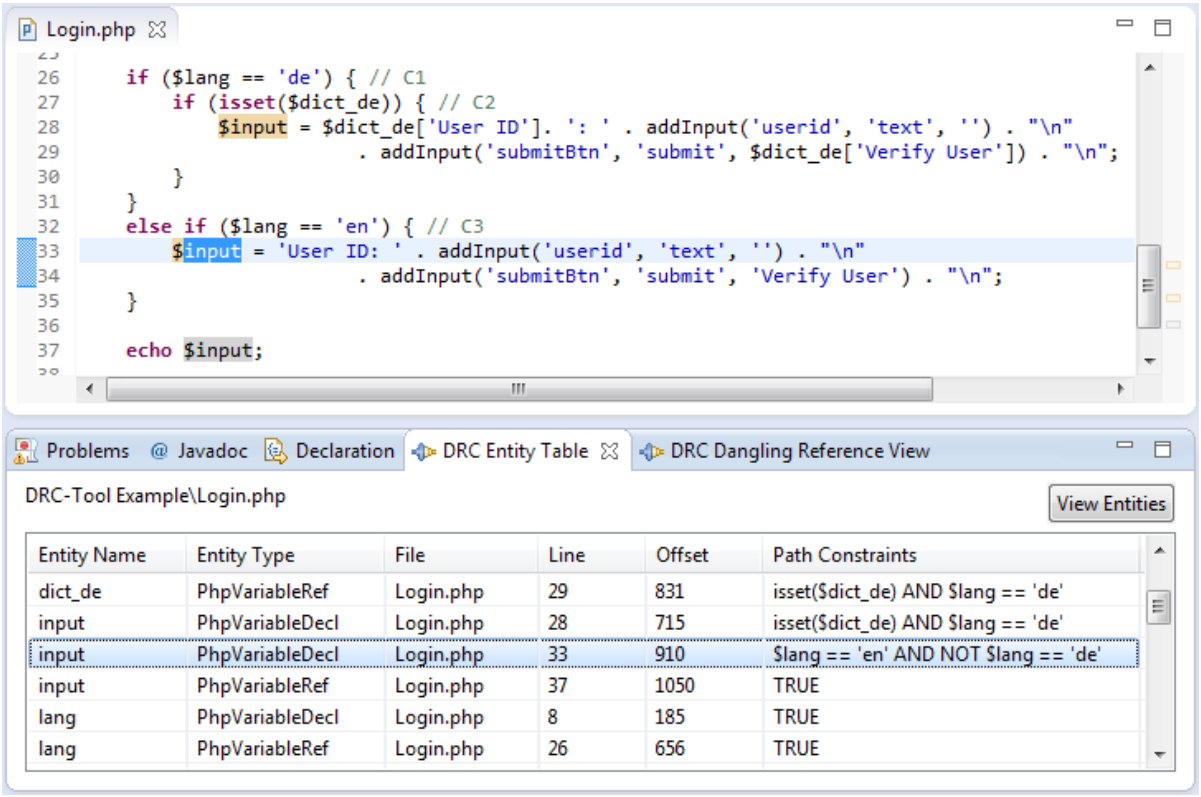


Figure 6.12 DRC's entity table

rations/references will be displayed on the right-hand-side panel (Reference List). If a reference cannot be matched to any declaration by names/types/constraints, DRC reports it as a dangling error. These dangling references are shown as a special entity group named "[Dangling References]" (Figure 6.13). As seen, DRC detects two dangling references in the file Login.php: the PHP variable $input on line 37 and the (embedded) JS reference to the HTML input userid on line 17 (not visible in Figure 6.13).

6.3.1.3 Case Studies

Figure 6.14 shows an example in which DRC detects that middleinitial (line 13) is an *SQL embedded* dangling reference, since middleinitial was missing from the database query (line 9).

Figure 6.15 displays DRC's detection result on SquirrelMail at revision 11,338. The

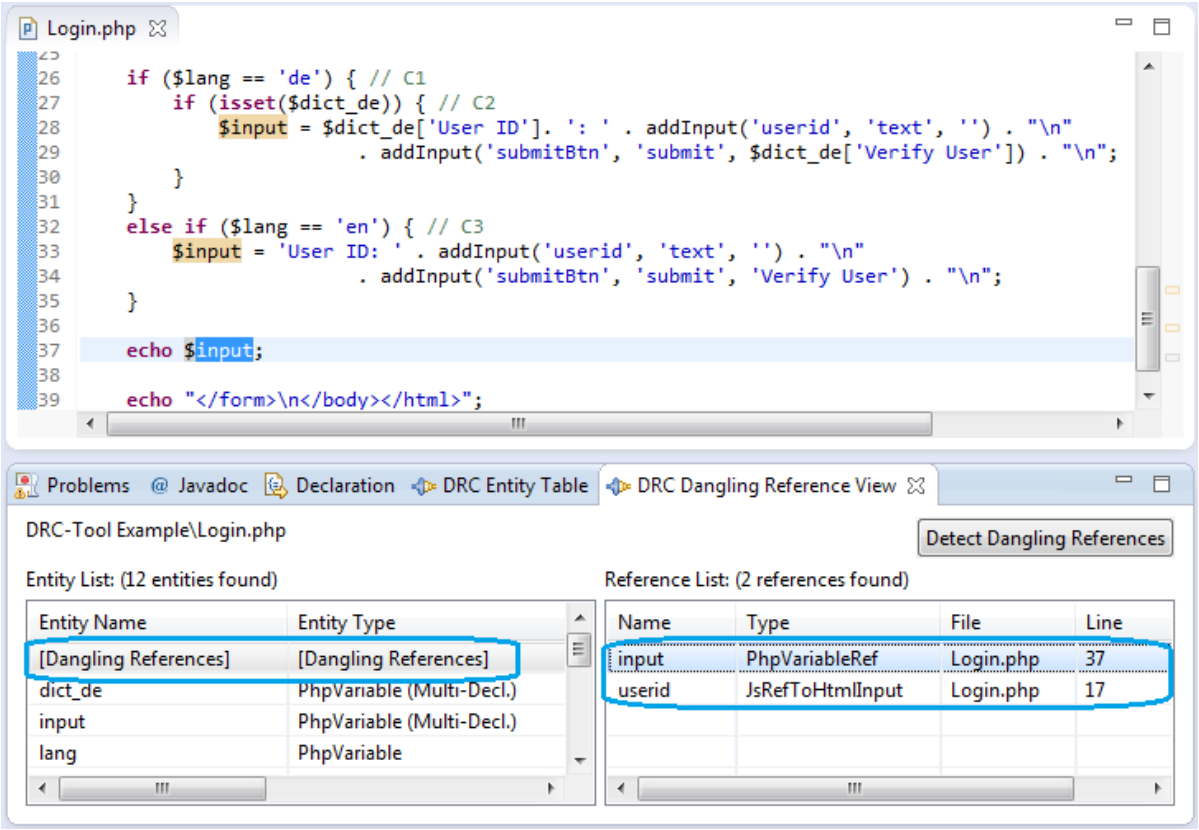


Figure 6.13 DRC's dangling reference detection

PHP variable $query is defined only when $response == 'NO' (line 397). In the else branch of the if statement on line 396 ($response == 'NO' evaluates to false), the variable $query is not defined, and thus its reference on line 405 is a dangling one. The commit log from the fix at the next revision confirms this error detected by DRC: *"$query is also used when $response != 'NO'. Fixed undefined notice error..."*

6.3.2 HTML Validation Error Detection

We propose *PhpSync* [105] (Figure 6.16), an bug-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 [76], an HTML validating/correcting tool to find any validation errors on the HTML page. If errors are detected, PhpSync leverages

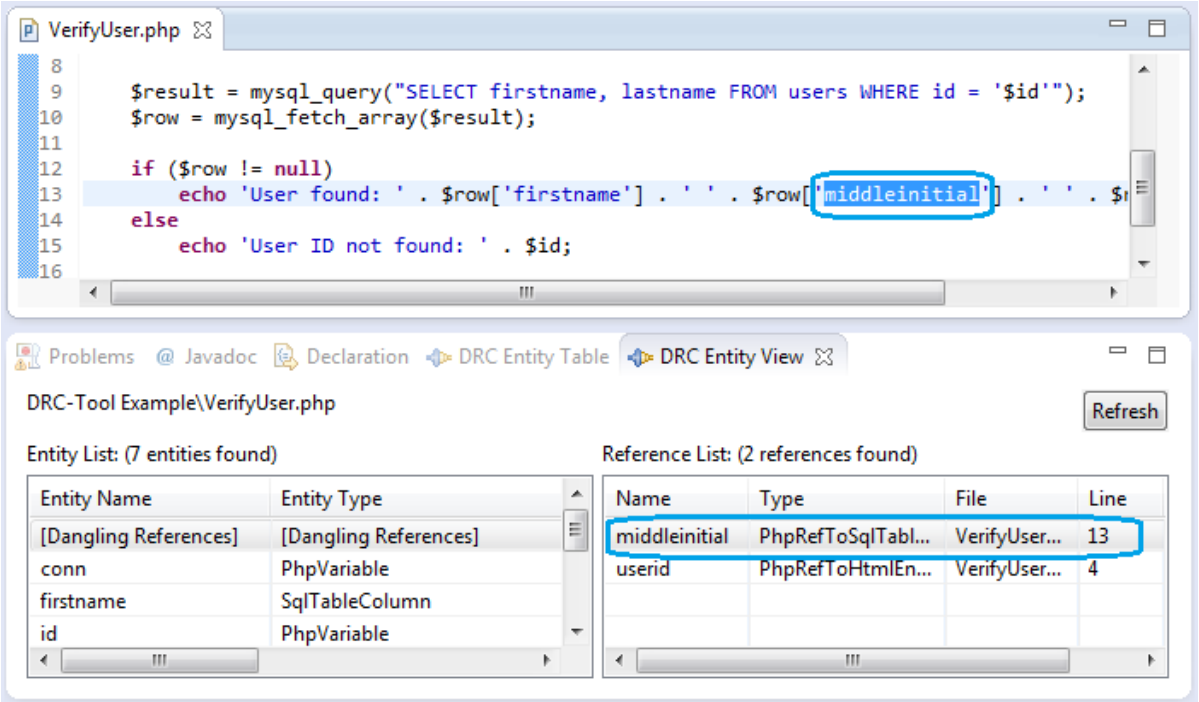


Figure 6.14 Embedded dangling reference detection in DRC

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 bug-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 inputs to our algorithm include a given HTML page C produced by a PHP page S. PhpSync uses Tidy [76] to check C for HTML validation errors. If errors are found, it uses Tidy to produce the corrected version C' of C.

Bug-Locating. There exist the cases in which Tidy is not able to provide the fixes [76]; however, it points out the buggy locations in the HTML page C. In such cases, for each error location in C, PhpSync uses our mapping algorithm (Section 5.3.3.2) to automatically locate the corresponding literal node(s) in the D-Model of S and then locate the PHP literal(s) in S.

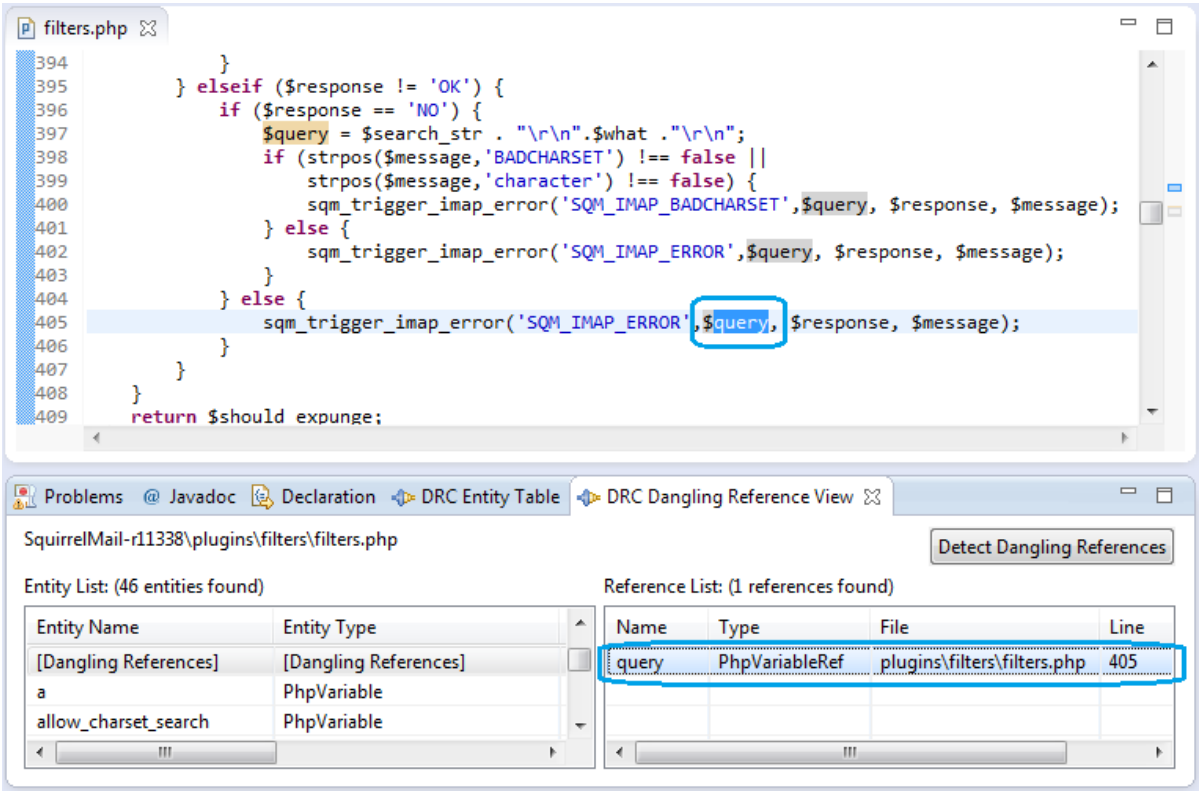


Figure 6.15 Dangling reference detected by DRC in SquirrelMail

Fix-Propagating. If Tidy can fix those errors, PhpSync will propagate those fixes through the mapping between S and C. Because Tidy does not provide the operations of the fixes but produces only the corrected version C', we 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.

6.4 Output Coverage Visualization for Output-Oriented Testing

Output coverage metrics (Section 5.3) summarize coverage in a number. To better understand output coverage, for example, in order to guide selection of additional test cases, we need to indicate which parts of the output have been covered, or, perhaps even more importantly, not covered. Highlighting covered and uncovered code fragments in IDEs as done by conventional code-coverage tools (e.g., EclEmma and Cobertura) or on

Figure 6.16 Bug-locating and fix-propagating for HTML validation errors to PHP server-side code

HTML source code addresses the wrong abstraction level for testers who are focused on the output and output-related quality criteria such as consistent font size and colors. Therefore, we borrow from coverage visualizations with background colors in IDEs, but apply them to the rendered output of a web page. To that end, we design a tool named *WebTest* that *displays the output universe in one single web page* and allows testers to visually *explore covered and uncovered parts* of the output universe, as initially exemplified in Figure 5.9c. Since the output universe can be significant in size and cover many alternative pages, we additionally develop more compact representations in which a user can interactively explore coverage in different parts of the output space. The testers can use WebTest to augment test cases or navigate and inspect the output universe directly to detect certain classes of presentation faults.

To display the output universe in a web browser and modify it to include branching decisions and coverage information, we need to understand the structure in the output universe, which can be represented by a VarDOM. Therefore, we encode the VarDOM

into a plain HTML page that can be read by a web browser to display on a web page. Since the VarDOM is an extended version of the DOM with the introduction of Condition nodes and symbolic values, we encode the VarDOM and corresponding coverage information into HTML as follows:

- *DOM nodes in the VarDOM* are directly rendered without modification: We simply display the corresponding HTML element and recursively displaying its child nodes (if any) in HTML format. We optionally show some key elements as *<html>* and *<title>* verbatim, as done in many WYSIWYG HTML editors to emphasize the structure of the page.

- *For a Condition node*, we make the decision explicit and show all alternatives together with their constraints in a single page. For example, in Figure 5.9c, we show both alternatives "Welcome $_GET['user']!" and "Please log in first." and their corresponding constraints. We explore an enhanced visualization showing only one alternative at a time below.

- *Symbolic values* are rendered using the original PHP code that represents the symbolic values. For instance, in Figure 5.9c, the symbolic value is shown as $_GET['user'].

- *Coverage information* is encoded as background colors, similar to IDE tools for code coverage. We highlight the parts that are entirely covered or uncovered by a given test suite in green and red background colors (injected by manipulating CSS attributes). Information about different coverage metrics is shown at the bottom part of the visualized web page (in Figure 5.9c, we show Cov_{str} only).

With this visual encoding of coverage information in the web output, our tool WebTest serves two key purposes: First, it allows testers to *explore different alternatives on different parts* of the web page. Note that our encoding algorithm maintains the original

layout and format of concrete web pages as far as possible so that the testers can visualize the actual web pages and quickly spot any presentation errors such as font size or spelling errors. Second, it allows testers to know *how well a test suite covers the output universe* and *explore uncovered scenarios or uncovered parts* of the web page.

Enhancement. Even with symbolic values, the output universe of a web page can be large, such that showing all alternatives side by side on a single page can lead to huge pages. Even in the relatively small SchoolMate system used in our evaluation, a rendering of all alternatives fills 59 printed pages. To address scalability and avoid overwhelming users, we enhance our original user interface while maintaining its key functionality. Specifically, we dynamically show and hide the alternatives on the output universe in tabs such that only one alternative is shown for each decision at a time, but such that a user can interactively explore alternatives.

A consequence of hiding part of the output universe is that coverage may be harder to assess. Instead of a binary red/green decision showing whether a fragment is covered, we indicate relative coverage for the entire fragment (including potentially hidden fragments of inner decisions) with our coverage metrics. That is, if we view part of the page with content that has been covered but that includes hidden uncovered alternatives, we visualize the corresponding coverage metric, such as 90 % Cov_{str} in a sidebar.

We show a screenshot of our output from the SchoolMate system in Figure 6.17. The *main panel* (on the left) shows the concrete web page for a specific combination of output decisions that a tester is currently exploring. The *side panel* (on the right) shows the made (possibly nested) decisions and allows a tester to explore alternatives in the output universe. When the tester clicks on one of the buttons listing alternatives, WebTest switches to the selected alternative (dynamically switching with JS) and displays the corresponding web page on the left. Coverage information is shown next to each alternative being selected. The tester can then create additional test cases that explore areas with lower coverage. In our screenshot, a tester has selected the first alternative out of

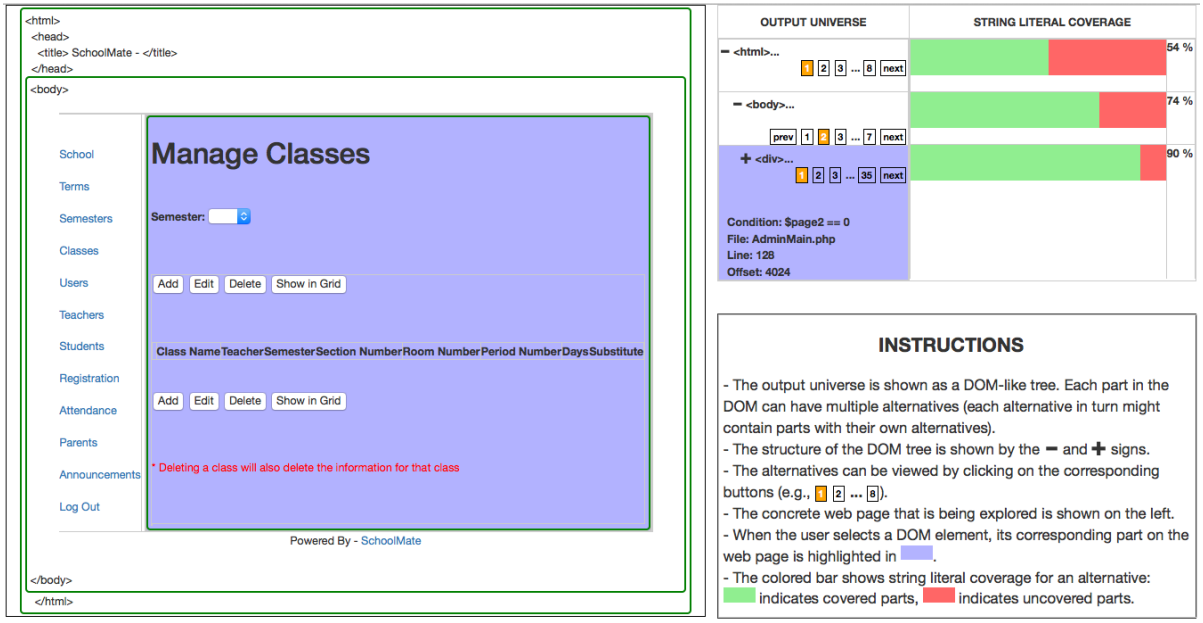


Figure 6.17 Screenshot of WebTest on SchoolMate-1.5.4

eight possible alternatives for the top-most decision (the other alternatives correspond to exception cases such as database connection error). The body contains another decision with seven alternatives out of which the second alternative is being explored (corresponding to the case that the user is logged in as administrator). Similarly, within that alternative, the first alternative out of the 35 alternatives is being shown (the logged in administrator explores the functionality to manage classes in this system). The sidebar additionally indicates that 90% of the output of this alternative has been explored.

CHAPTER 7. EMPIRICAL EVALUATION

In the following sections, we investigate the *accuracy, efficiency, and usefulness* of our cross-language program analysis techniques. Specifically, we evaluate the following:

- Call graphs for embedded client code (Section 5.1)
- Cross-language program slicing (Section 5.2)
- Dangling reference detection (Section 6.3.1)
- HTML validation error detection (Section 6.3.2)
- Output-oriented testing (Section 5.3)

7.1 Evaluation of Call Graphs for Embedded Client Code

Our approach to build call graphs has a number of sources of potential inaccuracies. While variability-aware parsing and call-graph building for HTML and CSS are conceptually sound, symbolic execution may not produce output for all string literals, parsing has limitations regarding symbolic values, and JS analysis uses potentially inaccurate reencodings, as we explained. In an evaluation with real-world PHP applications, we investigate the *practicality and accuracy* of our approach.

In addition, IDE support for navigation is especially useful if call-graph edges are nontrivial (e.g., developers need to search across files). Thus, we investigate the *complexity* of the created call graphs to characterize the *usefulness* of IDE support based on our tooling.

7.1.1 Experiment Setup

Table 7.1 Subject systems

Subject System	Version	Files	LOC
AddressBook (AB)	6.2.12	100	18,874
SchoolMate (SM)	1.5.4	63	8,183
TimeClock (TC)	1.04	69	23,403
UPB	2.2.7	395	104,640
WebChess (WC)	1.0.0	39	8,589

Table 7.2 Coverage on subject systems

System	Exec.	Output Size		Re-encoded JS		HTML	Cov. by Entries	
	Stmts	Chars	Conds	Stmts	Re-enc	Errors	Main	All
AB	1,546	15,480	195	223	28	12	32%	94%
SM	2,386	30,318	52	188	46	155	24%	92%
TC	1,311	15,157	99	128	36	22	11%	63%
UPB	5,175	35,587	711	876	0	30	6%	79%
WC	93	3,409	1	84	0	0	6%	97%

We collected five PHP web applications (Table 7.1) from sourceforge.net with various sizes and without heavy use of object-oriented constructs (also used in related work [105, 125]). For each system, we selected a main file, which might also recursively include other files, and ran our analysis on that file. When encountering HTML syntax errors during parsing (most often due to missing closing tags), we manually fixed them in the PHP code and report results after applying all fixes. All subject systems have multiple entry points (PHP files that can be called by a user), the call graphs of which can be merged, but due to the manual effort of fixing syntax errors, we considered only a single main file per system. The number of symbolically executed PHP statements, the size of the generated symbolic output, the size of JS code, and the number of fixed errors are listed in Table 7.2.

7.1.2 Practicality and Accuracy of Call Graphs

All call-graph computations completed within a few seconds (< 8 seconds on average, < 12 seconds in the worst case). Since this performance is acceptable for executing analyses in the background of an IDE, we did not perform additional rigorous performance measurements. Currently, we need to fix all HTML syntax errors before we can build call graphs, which is a positive side effect, but also a laborious endeavor. An error-recovering parser [32] or automated repair tools [125] could help with this step. Overall, our tool is easy to set up for a new project by pointing it to the main file(s).

Table 7.3 Complexity of call graphs

System	HTML Jumps			CSS Jumps		JS Jumps		Strings
	Total	xStr	xFiles	Total	xFiles	Total	xStr	on echo
AB	345	68%	2.3%	157	100%	7	85.7%	90%
SM	610	19%	6.2%	127	100%	33	100%	81%
TC	269	49%	4.5%	74	100%	2	0%	99%
UPB	386	49%	4.4%	136	100%	75	0%	44%
WC	40	63%	0%	20	100%	4	75%	97%
Tot/Avg	1,650	46%	3.5%	514	100%	121	52%	79%

xStr/xFiles: Number of call-graph edges that cross strings/files
Avg: Geometric mean for non-zero relative numbers (percentages)

Precision. In Table 7.3, we list the number of call-graph edges detected in all subject systems. We manually checked the created call graphs for correctness. In small call graphs (< 100 edges), we inspected all call-graph edges; in large call graphs, we randomly sampled 50 edges each. We consider a call-graph edge as correct when it connects two nodes tracked to PHP string literals, and the nodes are actually in a call relationship. All 604 checked edges were correct, yielding a *precision* of **100%**.

Recall. In the absence of ground truth, *recall* is more challenging to measure. We approach recall with three proxy metrics—(1) *coverage*, (2) *symbolic discipline*, and (3) *reencoding losses*—addressing the three sources of inaccuracies in our approach.

1. First, we can cover only literals that are output as part of an execution of the PHP code (string literals in dead code are never printed and we cannot build call graphs for them). In addition, due to limitations of our symbolic-execution engine (see Section 3), we may not cover every possible execution path or may not be able to track some string literals to their output. To characterize the potential loss of call-graph nodes, we measure *coverage* as the ratio between the number of output characters that are covered by symbolic execution and the total number of all output characters in the project. As a heuristic to ignore string literals that are not output, such as array index 'method' in $_GET['method'], we only consider literals that contain the representative HTML tag-opening character '<'. When executing the single main file, we cover 6% to 32% of all characters. When symbolically executing all entry points of the PHP code, we cover on average *84%* of all characters, (see the last column in Table 7.2). This shows that symbolic execution can achieve high coverage in our subject systems.

2. Second, we manually inspected all occurrences of symbolic values in the output of symbolic execution. We did not find a single case where a symbolic value was relevant for the structure of the VarDOM. That is, a symbolic value may add substructure but in no case was it required to provide the closing tag for a concrete opening tag or similar structural parts. This confirms our assumptions in Section 4.2 and makes inaccuracies due to symbolic values unlikely in our subject systems.

3. Third, while HTML and CSS analyses are sound with regard to a brute-force approach, our reencoding for JS analysis may lead to documents in which the used JS-call-graph tool WALA cannot find all call-graph edges. Note that we do not want to check the quality of WALA's call graphs but only to what degree our reencoding prevents discovering call-graph edges that WALA could discover without reencoding. Thus, we generated random configurations from the symbolic output (i.e., different selections for the #if decisions) and executed WALA on the generated code (without variability and without reencoding), comparing the resulting call-graph edges with the ones identified

when analyzing the entire configuration space with reencoding. We did not find a single case where reencoding resulted in missing call-graph edges compared to analyzing individual configurations.

Overall, the results and performance are practical and promising. In our subject systems, our tool yields a perfect precision. Investigating the sources of inaccuracies that lead to reduced recall shows that symbolic execution can cover 84% of string literals and that symbolic values during parsing and reencoding of JS are unproblematic. More details are in Table 7.2.

7.1.3 Complexity of Call Graphs

Besides accuracy, we investigated the potential benefit of such support. Using the created call graphs, we measure several characteristics to show the complexity of the underlying problem, suggesting the effort required when developers have no IDE support.

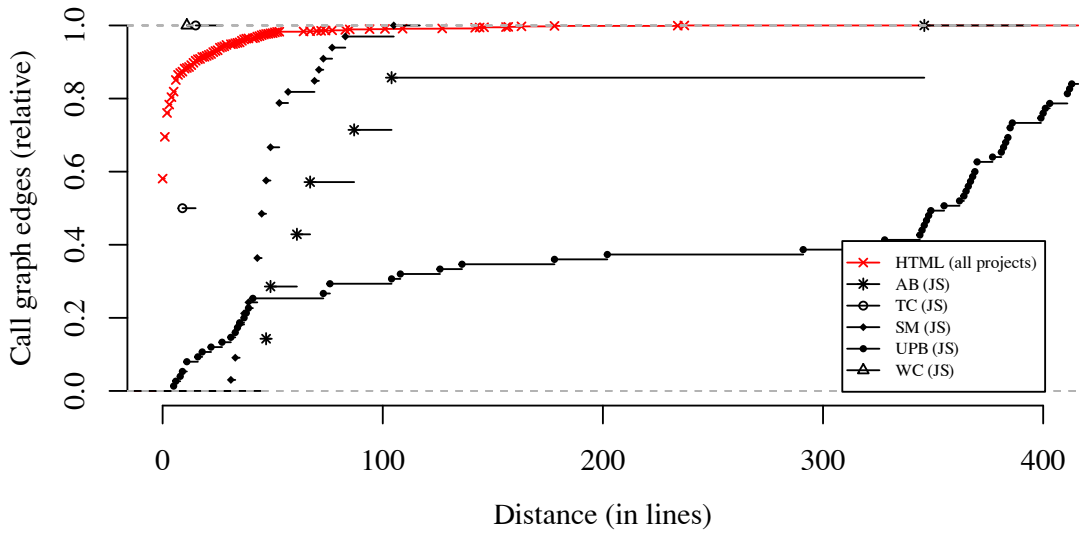


Figure 7.1 Cumulative distribution of distance of HTML and JS jumps (within the same file), depicting the percentage of jumps shorter than a given distance

First, we investigated the locality of call-graph edges. Call-graph edges are often nonlocal. 46% of all HTML call-graph edges on average and up to 100% of JS call-graph edges connect nodes in different string literals (Table 7.3). That is, the connected

elements are written in different parts of the server-side code, with some PHP code between them. While many call-graph edges for HTML are relatively local, with source and target in the same line or only few lines apart, about 8.3% are more than 20 lines apart and 3.5% are even in different PHP files. JS call-graph edges often span a larger distance, an average of 196 lines, but were mostly within the same PHP file. For CSS, even all call-graph edges connect nodes in different files since all CSS rules are written in separate files in our subject systems. Tool support is especially valuable for long jumps and jumps across files. Details are listed in Table 7.3 and Figure 7.1.

To characterize how difficult it is to track string literals through PHP code, when they are (re)assigned, concatenated, or part of other computations, we also tracked how many string literals are printed immediately or appear as inline HTML code in a PHP file. Again, we track only string literals containing the character '<'. We found that on average 21% of all such string literals are assigned to variables before they are eventually printed at a different location (Table 7.3). Our analysis can follow these string literals and create correct call graphs, while navigation without IDE support might not be straightforward.

Second, conditional jumping where one HTML tag is closed by alternative closing tags depending on the server-side execution is a challenge for tools and humans. Our solution with variability-aware parsing can correctly handle those cases and create corresponding conditional call-graph edges. We found two cases where the source of an HTML jump has multiple targets in AddressBook and TimeClock each, both similar to the <script> tag in our running example. Here, a developer may accidentally finish manual search before finding all relevant closing tags.

In addition, we found 21 cases in SchoolMate in which call-graph edges among nodes with the same name are disambiguated by their respective presence conditions, similar to the JS update call in our running example. Our results show that the produced client-side structure mostly aligns with the server-side execution, so that these cases are

relatively rare in practice. They also demonstrate that our more powerful infrastructure can provide accurate results in common as well as difficult cases.

Finally, a common approach for navigation in embedded client-side code is to use a global text search, especially for nonlocal jumps. A naive global text search for closing HTML tags, such as table, a, and form yields hundreds of results in dozens of files even in the smaller PHP projects. A global text search is only more promising for rare tags, such as html and body and uncommon JS variables and function names. For CSS, global text search is almost useless. A developer would not perform a *global* search in most cases and in many cases the corresponding jump target is only few lines away (see Figure 7.1), but nesting of HTML tags, common jumps across string literals, and occasional jumps over many lines of code and even across files (see Figure 7.1 and Table 7.3) show that a local search is also not a universal strategy. The relatively rare but possible case that a jump has alternative targets depending on the server-side execution (as the <script> tag in our running example), emphasizes that an incomplete local search may actually miss important targets potentially leading to inconsistencies. Overall, we conclude that text search can be an effective alternative in many common local cases but that a call graph can support navigation in many nontrivial cases quantified throughout our evaluation.

Threats to validity. Regarding external validity, we selected only a small sample of medium-sized subject systems and investigated only a single main file per project, due to the main bottle neck of manually fixing HTML syntax errors. While we cannot generalize over arbitrary PHP systems, all systems are real-world open-source applications developed by others and the results are consistent over all systems.

Regarding internal and construct validity, we used various proxy metrics to carefully characterize possible recall and usefulness measures. Since we do not have ground truth about what call graphs to expect, we decided to break down the evaluation into the three sources of inaccuracies. Implementation defects may also reduce precision and recall, but our tests and manual investigations did not reveal any issues. Instead of performing a

user study in which the navigation benefit may be buried in noise or over-exaggerated with artificial tasks or material, we decided to characterize usefulness by quantifying nontrivial call-graph edges in which developers could likely benefit from a tool. We expect a strong correlation with actual improvements in practice, but a usability study is still required.

7.2 Empirical Study on Cross-language Program Slicing

Program slicing tools are intended to help developers in various software maintenance tasks such as identifying the impact of a change. To evaluate a slicing approach, one could design a user study to show that slices are very difficult to manually identify or that developers could significantly benefit from a tool in complicated (favorable) cases. However, slicing in general has been shown to be useful in a number of applications [138, 152]. The more interesting question is how often such complicated cases occur. Therefore, we designed a study to quantify characteristics of data-flow dependencies and slices in existing web applications. Specifically, we are interested in how many entities are *embedded* within PHP strings, how many data-flow edges are *cross-language* or *cross-string*, how many slices cross languages and even web pages or require investigating embedded code fragments—all properties for which no current slicing tool is available. Although such complexity measures are only proxies for actual developer tasks, we argue that identifying a large set of complex data-flow dependencies or slices would demonstrate the benefit of automated slicing in dynamic web applications.

7.2.1 Experiment Setup

We implemented our cross-language program slicing technique in a tool called WebSlice (Section 5.2 and 6.2). To test the resulting data-flow graphs (and program slices) computed by WebSlice , we created 100 test cases for SchoolMate-1.5.4, a real-world web

application that we used in our study, covering all types of data flows. For data flows within PHP, we instrumented Quercus [118], an existing PHP interpreter, and dynamically tracked actual data-flow relations as a basis for our test oracles. For cross-language data flows and those within JS, we created the test cases manually. There are 20 test cases that include inter-page data flows; 2 of them failed because WebSlice included an infeasible edge (see last paragraph in Section 5.2.5). All test cases for other types of data-flow edges passed.

To answer our research questions, we collected from sourceforge.net five PHP web applications, also used in our previous study (Table 7.1). For each system, we automatically chose a set of page entries (i.e., PHP files that generate output containing an <html> tag) and ran WebSlice on those pages to create the data-flow graph for the entire system. To compute the slices, we considered each entity (a node in the data-flow graph) as a slicing criterion and calculated the program slice for the entity.

Table 7.4 Running time on subject systems

System	Entries	Exec. Stmts	Time
AB	17	25,713	10.0s
SM	1	2,942	5.2s
TC	32	26,388	13.3s
UPB	51	77,959	37.6s
WC	9	6,874	4.6s

The initial symbolic execution on all entries and construction of the data-flow graphs completed within a few seconds/entry for all systems (Tables 7.4). When the source code is changed, WebSlice needs to re-compute relevant page entries associated with the change only. This means that WebSlice can be run in the background of an IDE. Once the initial computations are finished, WebSlice can instantly show the program slice for any selected program point.

Table 7.5 Complexity of data-flow graph (nodes)

	PHP	Non-PHP Entities					
System	**Entities**	**Total**	**SQL**	**HTML**	**JS**	**Embed.**	**N-Echo**
AB	10,591	266	8	220	38	46	10
			3%	83%	14%	17%	4%
SM	4,935	2,402	404	729	1,269	2,402	452
			17%	30%	53%	100%	19%
TC	15,291	2,145	214	1,717	214	2,145	214
			10%	80%	10%	100%	10%
UPB	32,309	1,308	0*	1,160	148	1,191	447
			0%	89%	11%	91%	34%
WC	3,805	497	48	377	72	86	48
			10%	76%	14%	17%	10%
Total	**66,931**	**6,618**	**674**	**4,203**	**1,741**	**5,870**	**1,171**
			10%	**64%**	**26%**	**89%**	**18%**

N-Echo: Embedded entities that are not on echo/print *statements*
**There are 0 SQL entities in* UPB *since this system stores data in local files instead of an SQL database.*

7.2.2 Complexity of Data-Flow Graphs

We used our tool to investigate the complexity of data-flow graphs in dynamic web applications. Tables 7.5 and 7.6 show the complexity of data-flow graphs. Overall, developers would have to deal with a large number of SQL, HTML, and JS entities. There are a total of 4,203 HTML entities in all five systems, accounting for 64% of all non-PHP entities. There exist cases where developers would deal with up to 384 HTML entities *in a file* (e.g., in TimeClock) and up to 88 JS entities in a file (e.g., in SchoolMate). Especially, they must process as many as 89% of the non-PHP entities by examining embedded code in PHP strings (the remaining are directly inlined in PHP code). Moreover, not all embedded entities are printed directly on echo/print statements: 18% of them are assigned to variables, propagated through the program, and printed out at a different location, which makes it challenging to track the data flows without tool support. The edges in the data-flow graphs also demonstrate significant complexity. Out

Table 7.6 Complexity of data-flow graph (edges)

	Data-flow Edges					
System	**Total**	**xLang**	**xFile**	**xFunc**	**xString**	**xPage**
AB	13,406	416	2,538	3,043	474	356
		3%	19%	23%	4%	3%
SM	6,945	2,426	1,565	164	2,603	1,292
		35%	23%	2%	37%	19%
TC	16,490	655	3,493	46	937	332
		4%	21%	0%	6%	2%
UPB	38,186	4,983	6,986	3,121	5,470	4,886
		14%	19%	9%	15%	14%
WC	3,934	887	1,463	1,056	992	829
		23%	37%	27%	25%	21%
Total	**76,961**	**9,367**	**16,045**	**7,430**	**10,476**	**7,695**
		12%	**21%**	**10%**	**14%**	**10%**

xLang, xFile, xFunc, xString, xPage: Edges that cross languages, files, functions, strings, page entries

of 76,961 data-flow edges, 12% cross languages, 21% cross files, and 14% cross strings. This result shows that tool support would be useful in those cases.

7.2.3 Complexity of Program Slices

Table 7.7 Complexity of program slices

System	**Slices**	**Size**	**Len**	**xLang**	**xFile**	**xFunc**	**xString**	**xPage**
AB	6,827	6	5	287	2,330	3,202	344	243
SM	4,185	5	4	1,518	1,519	917	1,735	890
TC	9,145	6	4	1,193	2,007	643	1,378	224
UPB	17,906	7	5	795	7,904	8,386	1,236	681
WC	2,607	4	3	312	1,557	1,517	408	265
Tot/Avg	**40,670**	**5.6**	**4.2**	**4,105**	**15,317**	**14,665**	**5,101**	**2,303**
				10%	**38%**	**36%**	**13%**	**6%**

Size, Len: Median size and length of a slice

xLang, xFile, xFunc, xString, xPage: Slices that cross languages, files, functions, strings, page entries

Table 7.7 shows complexity metrics for slices (we exclude those that have only one entity since the entity is at the end of data flows).

Size of a slice. We compute the medians of sizes and lengths of slices and calculate their averages. On average, a developer would need to deal with a slice involving 5.6 entities and having a length of 4.2 (the longest path in the data-flow graph starting from a slicing criterion). 10% of the slices involve more than 40 entities (not shown in Table 7.7), which would be nontrivial to identify manually.

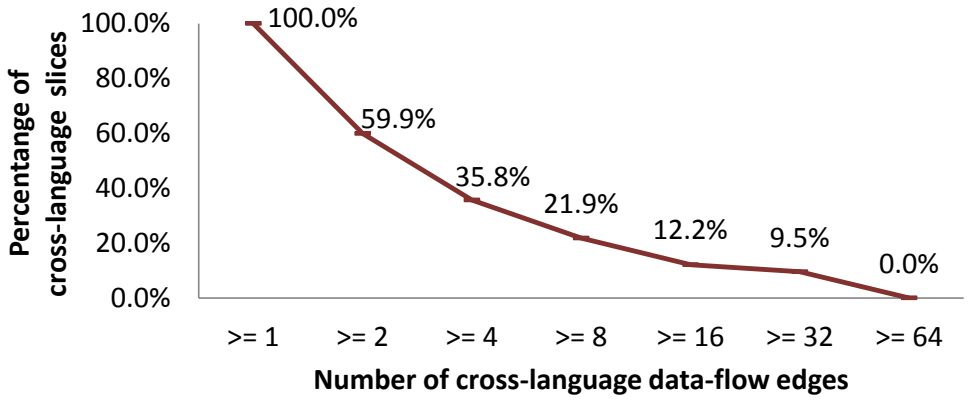


Figure 7.2 Cross-language data flows in a cross-language program slice

Cross-language data flows in a slice. Importantly, many of the slices are cross-language (in all five systems, 4,105 slices contain at least one cross-language data-flow edge). As shown in Figure 7.2, 35.8% of those slices have at least 4 cross-language data-flow edges, and 9.5% have at least 32 cross-language edges.

Cross-location data flows in a slice. Many slices are also often cross-location: 38% of the slices cross files, 36% cross functions, and 13% cross string fragments.

7.2.4 Discussion

Implications. The high complexity of the data-flow graphs and program slices shows that in real-world web applications, manually inspecting a program slice can be challenging, and developers would likely benefit from program slicing tool support.

Threats to validity. Regarding external validity, we used only a small set of medium-sized subject systems due to our limited support for PHP object-oriented con-

structs. Regarding construct validity, we used complexity as a proxy metric to show the usefulness of our program-slicing technique.

7.3 Evaluation of Dangling Reference Detection

This section presents our empirical studies to evaluate the accuracy of DRC, our tool for detecting dangling PHP and embedded references (Section 6.3.1).

7.3.1 Experiment Setup

Table 7.8 Subject systems and reported dangling references

System	History	Cand.	Dangling	PHP		Embed	
	Start	Rev	Rev	Ent	Ref	Ent	Ref
BeehiveForum	04/'02	173	16	12	18	6	6
ImpressCMS	12/'07	65	14	15	19	0	0
MRBS	05/'00	26	13	18	29	0	0
PHP-Fusion	03/'08	42	14	15	19	5	7
PhpWiki	06/'00	37	14	13	21	1	1
SquirrelMail	11/'99	47	17	21	23	0	0
TikiWiki	10/'02	87	15	13	17	3	3
All		477	103	107	146	15	17

We selected as our subject systems seven PHP-based dynamic web applications from sourceforge.net (Table 7.8) with high popularity rates and long histories. Column History Start shows their starting dates. To identify the *candidate revisions* (column Cand. Rev) that potentially contain the fixes for dangling bugs, we wrote a program to analyze the commit logs of all revisions (up to 07/1/12), and selected the ones containing the keywords such as "dangling", "dangled", "undefined", and "undeclared". To remove false positives, we then manually investigated all 477 revisions and collected those that actually contained a fix to at least one dangling reference error (column Dangling Rev). We used them as the oracle.

The last 4 columns show the numbers of cases that occurred in PHP code (PHP) and embedded code (Embed). For each type, columns Ent and Ref display the number of entities (variables/functions) having dangling references and the number of the dangling references, respectively. (One entity might have multiple references to it.) As seen in Table 7.8, 103 revisions were reported as the fixes for at least one dangling bug. There were 163 dangling references, with 146 and 17 being PHP and embedded ones.

We ran DRC on those systems and manually checked the results. If the detected dangling cases are covered those in the oracle, we counted them as correct ones. If DRC reported a dangling case that was not in the oracle, we manually verified if it is a truly incorrect case or a newly discovered one (not yet reported). We then computed precision and recall. *Precision* is the ratio between the number of correctly detected cases over the total number of detected ones. *Recall* is the ratio between the number of correctly detected ones over the number of cases.

7.3.2 Accuracy of Dangling Reference Detection

Table 7.9 DRC's detection accuracy

System	Corr	Incor	Miss	Pre%	Rec%	New
BeehiveForum	16:22	5:12	4:4	76%:65%	80%:85%	2:2
ImpressCMS	19:25	5:12	2:2	79%:68%	90%:93%	6:8
MRBS	37:50	7:14	3:5	84%:78%	93%:91%	22:26
PHP-Fusion	29:51	9:23	0:0	76%:69%	100%:100%	9:25
PhpWiki	13:24	2:6	4:5	87%:80%	76%:83%	3:7
SquirrelMail	21:26	6:8	4:4	78%:76%	84%:87%	4:7
TikiWiki	16:23	6:16	5:5	73%:59%	76%:82%	5:8
All	151:221	40:91	22:25	79%:71%	86%:89%	51:83

Table 7.9 shows the results. Columns Corr, Incor, and Miss show the number of dangling cases that were correctly/incorrectly detected, and missed by DRC. Since an entity might have multiple references, in each table cell, two numbers are reported: the first one is the number of dangling entities and the second is the number of dangling references.

As seen, DRC is accurate in dangling reference detection with an average of 89% recall (up to 100%) and 71% precision (up to 80%). Detection precision for dangling entities is higher. Interestingly, DRC discovered 83 not-yet reported cases (column New). We manually verified them as correct detection cases. DRC is also efficient. Detection time is typically less than two seconds per PHP file. Thus, our solution for matching constraints in the detection algorithm is practical. Let us present a few correctly detected cases by DRC next.

7.3.3 Case Studies

7.3.3.1 PHP Dangling References in MRBS at Revision 590

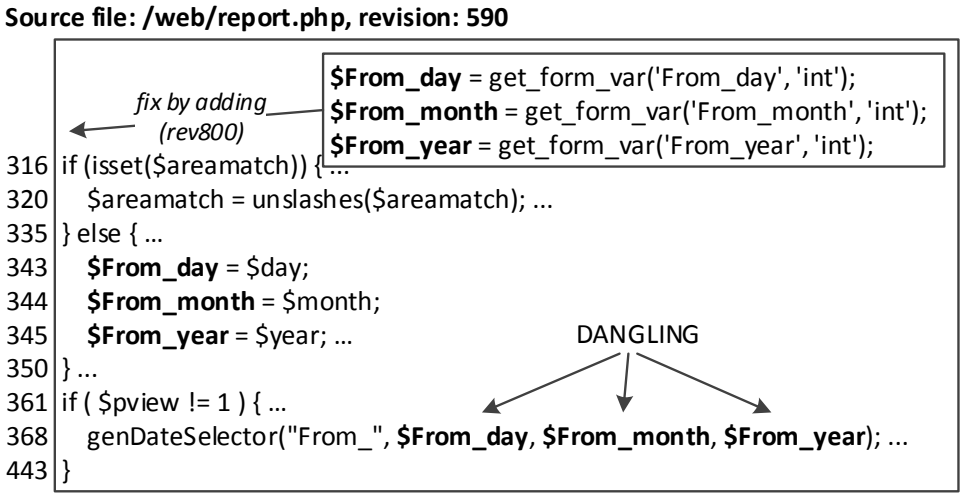


Figure 7.3 Newly found PHP dangling references in MRBS at revision 590

In Figure 7.3, in addition to detecting the dangling reference $typematch (not shown) reported in the commit log, DRC also found three other dangling ones that **were not reported at revision 590**: the three references to $From_day, $From_month, and $From_year on line 368 are dangling since they are initialized only in the else branch of the if statement on line 316. As shown in the fix, this was corrected 210 revisions later

by adding the variables' initializations before line 316. Thus, DRC could have helped developers detect those dangling errors early.

7.3.3.2 PHP Dangling Reference in ImpressCMS at Revision 3883

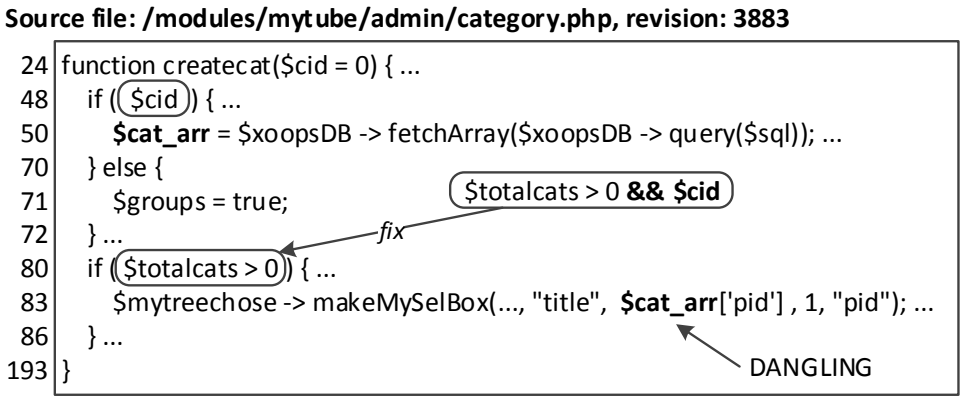


Figure 7.4 PHP dangling reference $cat_arr in ImpressCMS at line 83

In Figure 7.4, since the condition where $cat_arr is defined (line 48) and the condition where $cat_arr is accessed (line 80) do not match, there exists an execution where the variable $cat_arr is undefined (i.e., $cid=F and $totalcats > 0). As shown in the fix, developers added the check on $cid to the condition at line 80 so that $cat_arr is guaranteed to have been initialized whenever it is accessed. DRC was able to detect this case.

Limitations. We analyzed the inaccurate cases and identified the following limitations. First, there exist cases where our symbolic execution could not resolve the name of an included file, leading to incorrect identification of dangling references. Second, DRC does not consider the order of declarations and references, which could potentially cause inaccuracy. Third, our algorithm to find a solution for constraints also causes inaccuracy. Our symbolic execution to approximate the output created some missed embedded entities. Finally, DRC cannot handle the cases of eval to generate a portion of code.

7.4 Evaluation of HTML Validation Error Detection

This section presents our empirical evaluation on PhpSync, our bug-locating and fix-propagating tool for HTML validation errors in PHP-based web applications (Section 6.3.2). 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.

7.4.1 Experiment Setup

Table 7.10 Subject systems and D-Models

Subject Systems			D-Models			
Name	**Files**	**KLOCs**	**ExFiles**	**Nodes**	**Control**	**Time**
SchoolMate-1.5.4	63	8	60	5357	885	0.6s
TimeClock-1.4	69	23	7	886	101	0.1s
WebERP-4.0.2	654	220	16	59338	14841	2.8s
UPB-2.2.7	395	105	6	8383	1621	0.7s
AddressBook-6.2.12	100	19	10	1789	320	0.3s
Manhali-1.3.2	299	52	17	7952	2353	0.6s

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 7.10). 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.

7.4.2 Accuracy of Client Code and Server Code Mapping

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. Given an HTML test page, PhpSync divides its HTML contents into several text fragments and maps each fragment into the PHP literals/variables. 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* 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 other cases, we verified the mapping from f to l by the program semantics.

Table 7.11 Mapping and fixing result on SchoolMate-1.5.4

#	Navi Steps	Mapping							Fix-Propagating		
		Fragments				Characters (×1000)			Err.	Tidy	Php
		All	Auto	Man	Corr.	All	Corr.	Acc.			Sync
1	Login	15	12	3	15	7.0	7.0	100%	38	4	4
2	School	42	27	15	42	11.8	11.8	100%	50	19	19
3	Terms	45	31	14	43	12.2	12.1	99%	52	22	22
4	Semesters	51	35	16	49	12.5	12.4	99%	50	20	20
5	Classes	96	64	32	96	12.9	12.9	100%	57	27	27
6	Users	87	58	29	85	13.1	12.9	99%	64	34	34
7	Teachers	56	38	18	56	12.0	12.0	100%	50	20	20
8	Students	105	68	37	105	13.0	13.0	100%	50	20	20
9	Registration	102	69	33	102	12.4	12.4	100%	49	19	19
10	Attendance	73	50	23	72	11.8	11.6	99%	50	20	20
11	Parents	68	44	24	68	12.1	12.1	100%	50	20	20
12	Announce	45	31	14	43	12.3	12.2	99%	50	20	20
13	Terms/Add	19	15	4	19	10.3	10.3	100%	47	18	18
14	Terms/Edit	27	19	8	27	10.0	10.0	100%	47	18	18
15	Sem./Add	30	22	8	30	10.3	10.3	100%	47	18	18
16	Sem./Edit	43	30	13	43	10.4	10.4	100%	47	18	18
17	Classes/Add	47	32	15	47	11.0	11.0	100%	47	18	18
18	Classes/Edit	42	30	12	40	10.9	10.7	98%	47	18	18
19	Classes/Grid	22	17	5	22	9.8	9.8	100%	53	20	20
20	Users/Add	19	15	4	19	10.6	10.6	100%	48	19	19
21	Users/Edit	26	18	8	25	10.6	10.3	98%	48	19	19
		1060	725	335	1048	236.9	235.8	99.5%	1041	411	411

In Table 7.11, column Mapping shows the result on SchoolMate-1.5.4. 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.

Table 7.12 Mapping and fixing result on all subject systems

		Mapping									Fix-Propagating					
System	Test Pages	Fragments All	Auto	Man	Corr.	Characters (×1000) All	Corr.	Acc.	Complexity Files	Time	Err.	Tidy	Php Sync	Miss	Acc.	Time
SchoolMate-1.5.4	21	1060	725	335	1048	236.9	235.8	99.5%	6	4.4s	1041	411	411	0	100.0%	< 0.2s
TimeClock-1.4	14	2511	2103	408	2484	164.1	162.7	99.1%	5	1.2s	422	136	136	0	100.0%	< 0.2s
WebERP-4.0.2	10	2736	1910	826	2564	78.5	75.8	96.7%	5	8.0s	284	188	176	12	93.6%	< 1.0s
UPB-2.2.7	10	1234	866	368	1191	56.7	53.9	95.0%	4	0.6s	129	49	47	2	95.9%	< 0.2s
AddressBook-6.2.12	10	1853	1341	512	1841	78.9	77.5	98.1%	8	0.3s	64	51	48	3	94.1%	< 0.2s
Manhali-1.3.2	10	3717	2616	1101	3610	115.1	105.9	92.0%	9	3.8s	607	189	164	25	86.8%	< 0.2s

Column Mapping of Table 7.12 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.

7.4.3 Accuracy of Fix Propagation from Client Code 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 fixing changes between C and C^* and propagate them to fix the PHP code S into S^*. Then, we executed the fixed PHP code S^* and followed the same recorded navigation steps to produce the new HTML page C^+. Tidy was used to check on C^+ for validation errors again. After that, the lists of errors that Tidy had fixed ($C \rightarrow C^*$) and PhpSync had fixed via fix-propagation ($C \rightarrow C^+$) were automatically compared to determine how well PhpSync propagated those fixes. *Accuracy* is measured as the ratio between the number of correctly propagated fixes over the total propagated fixes. For the cases that Tidy uncovered validation errors but could not fix, one could use CSMap to auto-locate the erroneous PHP code.

The columns under Fix-Propagating in Tables 7.11 and 7.12 display the fix-propagation results. Column 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. Importantly, it did not introduce any new validation error.

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.

7.5 Empirical Study on Output-Oriented Testing

In this study, we are interested in whether or not our proposed output coverage metrics can serve as a useful metric for testers who are focused on testing the output of a web application. We call bugs that occur during server-side execution such as undefined variables and missing parameters as *code-related bugs*, and defects that manifest in the output—such as HTML validation errors, layout issues, and spelling errors—as *output-related bugs*. In a nutshell, we want to know whether output-coverage metrics are good indicators of test suites that are effective for detecting output-related bugs and whether they improve over classic code-coverage metrics for this class of bugs. Specifically, we will answer the following research questions:

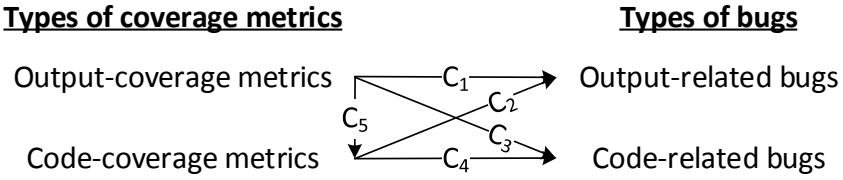
RQ1. *Do output-coverage metrics of a test suite correlate well with the number of detected output-related bugs?*

RQ2. *Do output-coverage metrics correlate with code-coverage metrics?*

RQ3. *Do output-coverage metrics improve over existing code-coverage metrics in detecting output-related bugs?*

RQ4. *Are traditional code-coverage metrics still better than output-coverage metrics for detecting code-related bugs?*

In other words, these questions investigate the correlations among the two categories of coverage metrics and the two types of bugs (namely C_1–C_5 as illustrated below). While RQ1 focuses on C_1 and RQ2 focuses on C_5, RQ3 is aimed at comparing C_1 and C_2, and RQ4 is aimed at comparing C_3 and C_4.



7.5.1 Experiment Setup

Table 7.13 Subject systems

Subject system		Size		Test
Name	**Version**	**Files**	**LOC**	**pool**
AddressBook (AB)	6.2.12	100	18,874	75
SchoolMate (SM)	1.5.4	63	8,183	67
TimeClock (TC)	1.04	69	23,403	63
UPB	2.2.7	395	104,640	67
WebChess (WC)	1.0.0	39	8,589	52
OsCommerce (OC)	2.3.4	787	91,482	92
WordPress (WP)	4.3.1	793	342,097	65

To answer those questions, we analyzed a corpus of seven open-source PHP web applications (Table 7.13), five of which were used in our previous study (Table 7.1); we further selected two large, popular systems for our study (OsCommerce and WordPress).

Computing output coverage and code coverage. The first step is to run symbolic execution to compute the output universe of each system. For test case generation, we use the web crawler Crawljax [31] to generate a *test pool* of test cases (column Test pool in Table 7.13). Among thousands of generated tests, we remove redundant test cases producing the same outputs. We then compute output-coverage metrics (Cov_{str} and Cov_{dec}) and code-coverage metrics (statement and branch coverage) for individual test suites, each containing a set of test cases in the test pool.

We repeat this process for each of the first five subject systems. For the last two systems, since our symbolic execution has not been well tested on these applications, we do not compute Cov_{dec}, and we approximate Cov_{str} with the total length of string literals that a test suite covers in the server-side program (as opposed to string literals covered in the output universe). We compute this alternative coverage directly using the origin information of string values recorded by our dynamic instrumentation during a test run (Section 5.3.3.1).

Computing bug coverage. Given a test case or test suite, we count the number of bugs that it reveals. For *output-related bugs*, we focus on the two following kinds:

- *HTML validation bugs*, as detected by the validation tool JTidy [76].
- *Spelling errors*, as detected by a simple open-source Java spell checker Jazzy [68]. Jazzy may flag false positives, such as unknown tool names, but we count those as spelling errors nonetheless, as a tester would have to investigate those as well.

For *code-related bugs*, we enable all levels of error reporting in PHP (setting error_reporting(E_ALL)) and collect all PHP errors that are reported during a test run.

Comparing output coverage and code coverage. With each type of bug (either output-related or code-related), we compare the effectiveness of code coverage and output coverage in detecting those bugs with two different strategies:

- *Random test suites:* We generate random test suites of different sizes (100 test suites per size). Test suites of the same size may have different coverage on output/code and bugs. By comparing the (Pearson) *correlations* between a coverage metric and the number of detected bugs within test suites of the same size, we can judge the effectiveness of a metric. We repeat this measurement across all possible sizes (from size 1 to the size of the test pool) to observe the overall trend. We expect that output-coverage metrics are a better predictor for output-related bugs than code-coverage metrics.

- *Optimized test suites:* To simulate a developer optimizing a test suite toward a specific metric, we optimize the test suite with k tests such that it maximizes a coverage metric. Specially, we implemented a (deterministic) greedy algorithm that incrementally adds test cases to a test suite such that each added test case makes the resulting test suite achieve the highest coverage. We create such an optimized test suite for all possible sizes k and for each metric (Cov_{str} and statement coverage). We expect that selecting a test suite optimized for output coverage is more effective at finding output-related bugs than a test suite optimized for code coverage.

The two experiments assess complementary aspects. The first assesses the predictive power of a coverage metric, whereas the second assesses the effect of using a coverage metric as an optimization criterion when creating a test suite (and the effectiveness of this strategy at different sizes of the test suite).

7.5.2 Results with Random Test Suites

Table 7.14 show the results of our experiment with random test suites. (RQ1) For most systems, output coverage (Cov_{str} and Cov_{dec}) correlate well with bug coverage for both kinds of output-related bugs (HTML validation errors and spelling errors). However, Cov_{str} tends to correlate better with output-related bug coverage than Cov_{dec}

Table 7.14 Pearson correlations among output coverage, code coverage, and output-related bug coverage for random test suites. Correlations are computed for a set of test suites of the same size. A pair of values $m \pm s$ indicates the mean m and standard deviation s of correlation values across different sizes of test suites.

	Cov_{str} vs.	HTML Validation Errors			
System	**Stmt Cov**	Cov_{str}	**Stmt Cov**	Cov_{dec}	**Branch Cov**
AB	0.65 ± 0.14	0.76 ± 0.07	0.33 ± 0.12	0.81 ± 0.06	0.32 ± 0.11
SM	0.75 ± 0.08	0.76 ± 0.05	0.61 ± 0.10	0.58 ± 0.07	0.48 ± 0.15
TC	0.87 ± 0.06	0.82 ± 0.05	0.83 ± 0.07	0.16 ± 0.10	0.57 ± 0.08
UPB	0.27 ± 0.18	0.34 ± 0.12	0.17 ± 0.11	-0.07 ± 0.09	0.15 ± 0.11
WC	0.51 ± 0.12	0.35 ± 0.24	0.30 ± 0.11	0.29 ± 0.23	0.27 ± 0.14
OC	0.80 ± 0.19	0.59 ± 0.13	0.41 ± 0.13	N/A	0.39 ± 0.14
WP	0.62 ± 0.07	0.48 ± 0.09	0.62 ± 0.07	N/A	0.58 ± 0.08

*N/A: Cov_{dec} is not available since we do not run symbolic execution to compute the output universe in these systems. For these systems, we approximate Cov_{str} as discussed in Section 7.5.1.

	Spelling Errors			
System	Cov_{str}	**Stmt Cov**	Cov_{dec}	**Branch Cov**
AB	0.56 ± 0.15	0.03 ± 0.12	0.56 ± 0.15	0.00 ± 0.11
SM	0.41 ± 0.17	0.09 ± 0.13	0.19 ± 0.14	0.05 ± 0.12
TC	0.48 ± 0.08	0.52 ± 0.08	0.33 ± 0.10	0.49 ± 0.08
UPB	0.52 ± 0.10	0.24 ± 0.18	0.25 ± 0.13	0.20 ± 0.18
WC	0.93 ± 0.07	0.39 ± 0.16	0.56 ± 0.17	0.12 ± 0.12
OC	0.78 ± 0.17	0.77 ± 0.27	N/A	0.75 ± 0.26
WP	0.85 ± 0.07	0.44 ± 0.09	N/A	0.32 ± 0.11

since Cov_{str} better measures "the amount of output" that is covered by a test suite. This suggests a test selection strategy which optimizes a test suite based on Cov_{str}.

(RQ2) As shown in the second column, Cov_{str} does not strongly correlate with statement coverage. This implies that output coverage and code coverage can be used for different purposes, and classic code coverage metrics might not be suitable for measuring coverage relating to the output. (RQ3) In particular, Table 7.14 indicates that while code coverage metrics also have some correlation with output-related bug coverage, their

Table 7.15 Correlations between output coverage and code coverage with code-related bug coverage for random test suites (similar to Table 7.14 but for PHP errors)

	PHP Errors			
System	Cov_{str}	**Stmt Cov**	Cov_{dec}	**Branch Cov**
AB	0.45 ± 0.13	0.58 ± 0.13	0.43 ± 0.14	0.52 ± 0.14
SM	0.48 ± 0.28	0.72 ± 0.15	0.58 ± 0.21	0.80 ± 0.12
TC	N/A			
UPB	0.13 ± 0.09	0.27 ± 0.11	0.21 ± 0.13	0.28 ± 0.11
WC	0.48 ± 0.26	0.40 ± 0.14	0.37 ± 0.27	0.22 ± 0.14
OC	N/A			
WP	N/A			

*N/A: There are no PHP errors reported.

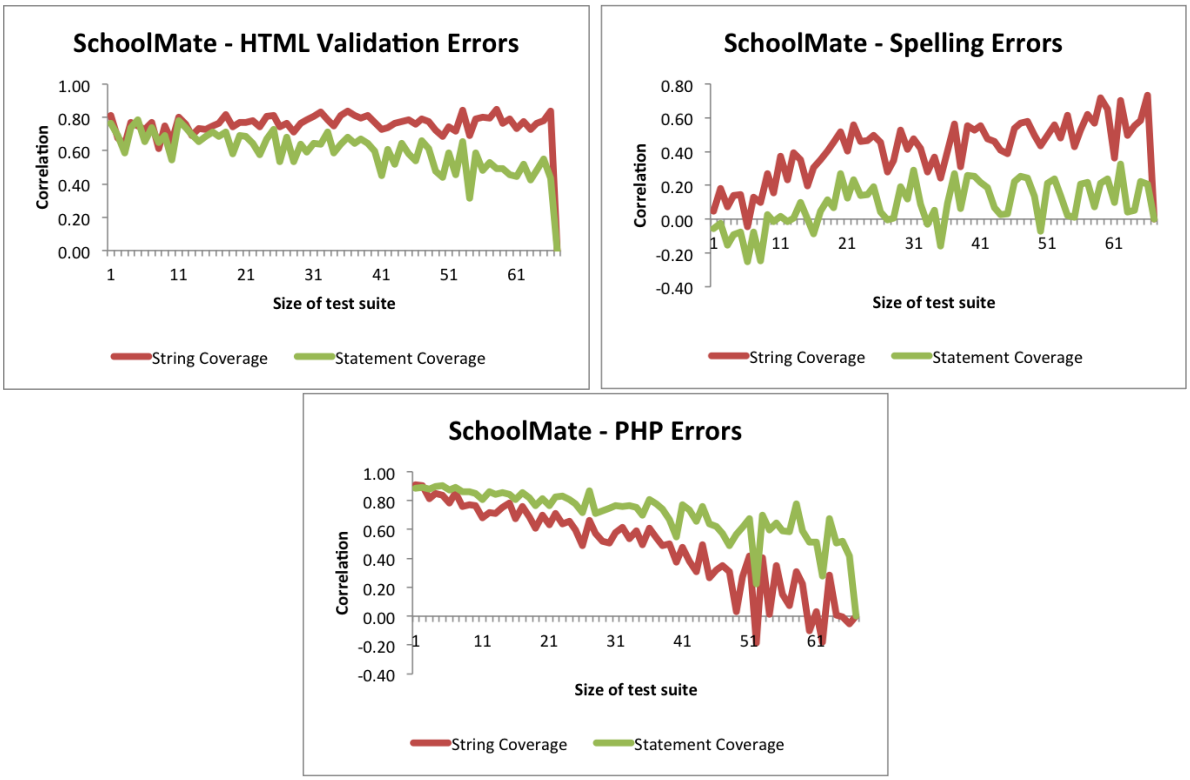


Figure 7.5 Comparison of Cov_{str} and statement coverage for the first experiment in SchoolMate. (Note that for the largest test suites, the correlation is 0 since there is only one largest test suite.)

correlations are generally not as strong as the correlations between output coverage and output-related bug coverage. (RQ4) On the other hand, code coverage metrics have better correlations with code-related bug coverage (Table 7.15). Figure 7.5 depicts this trend for the SchoolMate system.

7.5.3 Results with Optimized Test Suites

Table 7.16 Comparison between test suites optimized for output coverage (T_o) and those optimized for code coverage (T_c). Given a type of coverage, a value in the table equals $(W - L)$, where W is the number of times that T_o has larger coverage than T_c and L is the number of times that T_o has smaller coverage than T_c.

	Types of Bug Coverage			Output/Code Cov	
System	**Validation**	**Spelling**	**PHP**	Cov_{str}	**Stmt Cov**
AB	8	11	-6	16	-31
SM	10	18	-50	46	-64
TC	42	43	N/A	52	-52
UPB	-4	14	-13	56	-66
WC	15	21	-28	23	-51
OC	31	1	N/A	15	-65
WP	-3	42	N/A	48	-62

*N/A: There are no PHP errors reported.

The results from Table 7.16 show that, across all systems, test suites that are optimized for output coverage tend to detect more output-related bugs than those that are optimized for code coverage. (The absolute values of the numbers in the table are less than the size of the test pool because when test suites are sufficiently large, all bugs are found, and in such cases test suites optimized towards different criteria all achieve the same coverage). On the other hand, test suites optimized for code coverage detect more PHP errors. These results are consistent with the observations for RQ3 and RQ4 in the previous experiment. We show this trend for the AddressBook system in Figure 7.6.

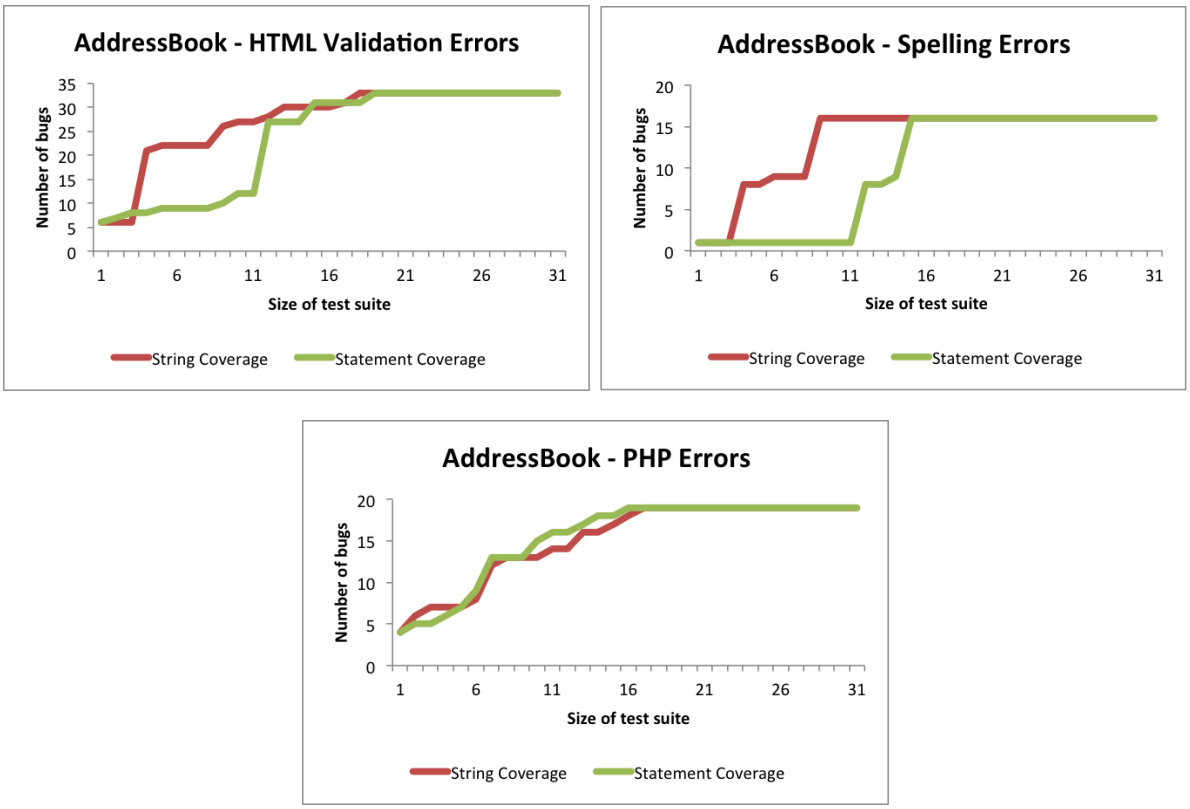


Figure 7.6 Comparison of Cov_{str} and statement coverage for the second experiment in AddressBook

To further see the effects of measuring output coverage as opposed to code coverage, we also compare the coverage of the existing optimized test suites with regard to their own coverage criteria (the last two columns in Table 7.16). (This corresponds to the potential of each type of coverage for detecting bugs, assuming an equal distribution of bugs on the output or code.) The results show that there are significant differences between output coverage and code coverage, which is again consistent with the results for RQ2 in the previous experiment.

Investigating the cases where output coverage performs better than code coverage in detecting output-related bugs, we found that many systems contain large pieces of code for purposes such as user credentials validation, numerical computations, or filesystem operations, which—when executed—do not contribute significantly to the output. In

such cases, a high coverage on code does not always translate into a high coverage on the output, and hence, a high coverage of output-related bugs.

7.5.4 Discussion

Our goal is to provide coverage metrics that are complementary to code coverage. The results from the previous experiments indicate that output coverage has equal or better performance than classic code coverage in terms of detecting output-related bugs. With regard to complexity, the initial symbolic execution ran within seconds for each of the first five systems, and our output coverage computation completed for less than a second per test case. While our symbolic-execution engine has limited support for object-oriented code (Section 3), issues with larger systems (e.g., WordPress) and object-oriented code are engineering issues in supporting more language constructs, which do not affect our concepts. We currently do not handle JS and client-side processing as it is out of this paper's scope.

CHAPTER 8. CONCLUSION

With the goal of improving software quality and reliability, various methods have been introduced to aid developers in writing and maintaining software. Despite their increased popularity in supporting traditional software applications, those methods face a number of challenges when being applied to *multilingual, dynamic web applications*. Specifically, in a web application, the client-side code (in HTML, JS, and CSS) is routinely intermixed with the server-side code, embedded in scattered and incomplete string fragments, and there exist a potentially exponential number of client code variants, similar to a family C programs with #ifdefs. Due to these challenges, there is limited program analysis and IDE support for dynamic web applications in comparison to traditional applications.

We tackled those challenges by designing a two-phase approach using two key ideas. In the first phase, we transform a program with mixed server-side code and client-side code into one that contains only client-side code, and captures its variants via a representation resembling C code with #ifdefs. In the second phase, we reuse and adapt the state-of-the-art approaches in the analysis for programs written C code with #ifdefs to apply for the analysis on client-side code variants written in HTML, JS, and CSS.

Based on those key ideas, we introduced *an infrastructure for cross-language program analysis for dynamic web applications*. Specifically, we have developed the following research components. (The first component corresponds to the first phase, whereas the other components correspond to the second phase.)

- *An output-oriented symbolic execution engine* that approximates all possible outputs of a PHP-based web application [105, 102]. The result of symbolic execution

is the generated client-side code which possibly contains symbolic values and conditional values, represented by a D-Model [105].

- *A set of variability-aware parsers for conditional HTML, JS, and CSS code*, which are able to parse conditional symbolic output into a VarDOM that compactly represents all DOM variations [101].

- *Basic cross-language analyses*: Based on the VarDOM, we develop concepts, techniques, and tools (1) to build call graphs for embedded client code in different languages [101, 103], (2) to compute cross-language program slices [102], and (3) to support testing dynamic web applications via a novel test coverage criterion called output coverage.

- *A number of software development services for dynamic web applications*: We developed various tools and Eclipse plug-ins to support syntax highlighting, code completion, and code navigation for embedded code [101, 103], refactoring on embedded entities [106], cross-language program slicing [102], detecting dangling references [104, 107], detecting HTML validation errors [105], and visualizing output coverage for testing.

In summary, this thesis has made the following contributions:

1. **Concepts:** (1) the new concept of embedded client code in web code, (2) the notion of call graph for embedded client code, (3) the notion of program slices across different languages, and (4) the notion of different types of testers concerning different aspects of the software including its output, as well as a family of output coverage metrics of a test suite

2. **Representations:** The D-Model representation encoding different possible textual contents of the output and the VarDOM representation which compactly represents DOM variations generated from the server-side code

3. **Algorithms:** A systematic approach combining symbolic execution, variability-aware parsing, and variability-aware analysis (1) to build call graphs for embedded HTML, CSS, and JS code, (2) to compute cross-language data-flow relations and program slices, and (3) to compute output coverage metrics

4. **Tooling:** A toolchain that provides various kinds of software development support for dynamic web applications including IDE services, fault localization, bug detection, and testing

5. **Empirical studies:** Empirical evaluations on several real-world web applications to investigate the complexity of call graphs, data flows, program slices, the accuracy of bug detection, and the effectiveness of output coverage metrics, showing the analyses' accuracy, efficiency, and usefulness

Overall, the take-home message is that the cross-language program analysis infrastructure developed in this dissertation has made possible a wide range of web development services. Apart from the core analyses and services mentioned above, our framework is also designed to be extensible and adaptable. Web developers can not only make use of our existing tools but can also extend the framework to create new ones. For example, future work can make use of this framework to explore other important areas in web application development such as web security. In addition, one can apply similar techniques proposed in this thesis to other types of software applications with the same code-generation mechanism such as build code.

BIBLIOGRAPHY

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98. IEEE Computer Society, 2007.

[2] H. Agrawal and J. Horgan. Dynamic program slicing. *ACM SIGLAN Notices*, 25:246–256, 1990.

[3] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of ISSRE 1995*, 1995.

[4] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.*, 36(6):742–762, Nov. 2010.

[5] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 947–957. IEEE Press, 2012.

[6] M. Alkhalaf, S. R. Choudhary, M. Fazzini, T. Bultan, A. Orso, and C. Kruegel. Viewpoints: differential string analysis for discovering client- and server-side input validation inconsistencies. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 56–66, 2012.

[7] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

[8] N. Alshahwan and M. Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1345–1348. IEEE Press, 2012.

[9] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 181–192, New York, NY, USA, 2014. ACM.

[10] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[11] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4:326–345, 2005.

[12] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.

[13] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491. IEEE Computer Society, 2013.

[14] S. Apps. Defining static web apps. https://staticapps.org/articles/defining-static-web-apps/, 2016. Accessed: 2016-03-07.

[15] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580. ACM, 2011.

[16] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 265–274. ACM Press, 2010.

[17] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 261–272. ACM, 2008.

[18] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng.*, 36(4):474–494, 2010.

[19] L. Aversano, M. D. Penta, and I. D. Baxter. Handling preprocessor-conditioned declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 83–92. IEEE CS, 2002.

[20] I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. IEEE Computer Society, 2001.

[21] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, Jan. 1985.

[22] D. Binkley and K. Gallagher. Program slicing. *Journal of Advanced Computing*, 43:1–50, 1996.

[23] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2014.

[24] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Journal of Advanced Computing*, 62:105–178, 2004.

[25] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL^{LIFT}: Statically analyzing software product lines in minutes instead of years. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM Press, 2013.

[26] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 122–131. IEEE Computer Society, 2013.

[27] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

[28] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Inf. Soft. Technology*, 40(11-12):595–608, 1998.

[29] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 81–92. ACM Press, 2011.

[30] S. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold. Localizing SQL Faults in Database Applications. In *Proceedings of International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011.

[31] Crawljax. Crawljax website. http://crawljax.com/, 2015. Accessed: 2015-11-21.

[32] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Trans. Program. Lang. Syst.*, 34(4):15:1–15:50, 2012.

[33] A. de Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, WPC '96. IEEE Computer Society, 1996.

[34] G. A. Di Lucca and M. Di Penta. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, WSE '05, pages 87–94. IEEE Computer Society, 2005.

[35] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. Reverse engineering web applications: the ware approach. *J. Softw. Maint. Evol.*, 16:71–101, January 2004.

[36] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for erp professionals via program slicing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 97–108. ACM, 2008.

[37] S. Doğan, A. Betin-Can, and V. Garousi. Web application testing: A systematic literature review. *J. Syst. Softw.*, 91:174–201, May 2014.

[38] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Higher-Order and Symbolic Computation*, pages 105–118, 1999.

[39] ECMA. ECMAScript language specification - ECMA-262 edition 5.1. http://www.ecma-international.org/ecma-262/5.1/, 2015. Accessed: 2015-11-21.

[40] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.

[41] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 391–406. ACM Press, 2011.

[42] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.

[43] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 119–138. ACM Press, 2011.

[44] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, ICSE '13, pages 752–761. IEEE Press, 2013.

[45] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 379–392. ACM, 1995.

[46] M. Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[47] L. Frantzen, M. Las Nieves Huerta, Z. G. Kiss, and T. Wallet. Web services and formal methods. chapter On-The-Fly Model-Based Testing of Web Services with Jambition, pages 143–157. Springer-Verlag, Berlin, Heidelberg, 2009.

[48] K. Gallagher, D. Binkley, and M. Harman. Stop-list slicing. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 11–20. IEEE Computer Society, 2006.

[49] J. Garcia, D. Popescu, G. Safi, W. G. J. Halfond, and N. Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 367–377. ACM, 2013.

[50] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM Press, 2012.

[51] C. Girardi, F. Ricca, and P. Tonella. Web crawlers compared. *International Journal of Web Information Systems*, 2(2):85–94, 2006.

[52] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM Press, 2005.

[53] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510. ACM, 1975.

[54] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[55] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 174–183. ACM, 2005.

[56] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, WPC '97, pages 70–. IEEE Computer Society, 1997.

[57] M. Harman, S. Danicic, Y. Sivagurunathan, and D. Simpson. The next 700 slicing criteria. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*, 1996.

[58] M. Harman and K. Gallagher. Program slicing. *Inform. Softw. Technol.*, 40:577–582, 1998.

[59] M. Harman and R. Hierons. An overview of program slicing. *Softw. Focus*, 3:85–92, 2001.

[60] M. Harman, R. Hierons, C. Fox, S. Danicic, and J. Howroyd. Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01. IEEE Computer Society, 2001.

[61] P. Heidegger and P. Thiemann. Contract-driven testing of javascript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172, Berlin, Heidelberg, 2010. Springer-Verlag.

[62] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46. ACM, 1988.

[63] S. S. Huang and Y. Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 33(2):6:1–44, 2011.

[64] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, volume 3676, pages 309–326. Springer-Verlag, 2005.

[65] P. Hudak. Modular domain specific languages and tools. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 134–142. IEEE Computer Society, 1998.

[66] Humbug. Mutation testing framework for php. https://github.com/padraic/humbug, 2016. Accessed: 2016-03-07.

[67] JavaBDD. JavaBDD website. http://javabdd.sourceforge.net/, 2015. Accessed: 2015-11-21.

[68] Jazzy. Jazzy website. http://sourceforge.net/projects/jazzy/, 2015. Accessed: 2015-11-21.

[69] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 167–178. ACM, 2008.

[70] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 2011.

[71] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the International Static Analysis Symposium (SAS)*. Springer-Verlag, 2009.

[72] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 38–47. ACM, 2005.

[73] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282. ACM, 2005.

[74] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477. ACM, 2002.

[75] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 86–96. ACM Press, 1986.

[76] JTidy. Jtidy website. http://jtidy.sourceforge.net/, 2015. Accessed: 2015-11-21.

[77] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3):14:1–14:39, 2012.

[78] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824. ACM, 2011.

[79] L. C. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 444–463. ACM Press, 2010.

[80] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 199–209. IEEE Computer Society, 2009.

[81] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[82] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29:155–163, October 1988.

[83] Y.-F. Li, P. K. Das, and D. L. Dowe. Two decades of web application testing-a survey of recent advances. *Inf. Syst.*, 43(C):20–54, July 2014.

[84] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26. ACM, 2005.

[85] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM Press, 2013.

[86] S. Litvak, N. Dor, R. Bodik, N. Rinetzky, and M. Sagiv. Field-sensitive program dependence analysis. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 287–296. ACM, 2010.

[87] A. D. Lucia. Program slicing: Methods and applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE CS, 2001.

[88] J. Lyle and M. Weiser. Automatic bug location by program slicing. In *Proceedings of the ICCEA*, ICCEA'87, pages 877–883, 1987.

[89] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM Press, 2013.

[90] S. Mahajan and W. G. Halfond. Finding html presentation failures using image comparison techniques. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 91–96. ACM, 2014.

[91] S. Mahajan, B. Li, and W. G. J. Halfond. Root cause analysis for html presentation failures using search-based techniques. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, SBST 2014, pages 15–18. ACM, 2014.

[92] J. Maras, J. Carlson, and I. Crnkovic. Client-side web application slicing. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 504–507. IEEE Press, 2011.

[93] P. McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[94] A. Mesbah and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[95] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, Mar. 2012.

[96] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 67–78, New York, NY, USA, 2014. ACM.

[97] J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, Feb. 1963.

[98] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 432–441. ACM Press, 2005.

[99] M. Mirzaaghaei and A. Mesbah. Dom-based test adequacy criteria for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 71–81, New York, NY, USA, 2014. ACM.

[100] D. Nations. What is a web application? http://webtrends.about.com/od/webapplications/a/web_application.htm, 2014. Accessed: 2016-02-09.

[101] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 518–529, New York, NY, USA, 2014. ACM.

[102] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 369–380, New York, NY, USA, 2015. ACM.

[103] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Varis: IDE support for embedded client code in PHP web applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, pages 693–696, Piscataway, NJ, USA, 2015. IEEE Press.

[104] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Dangling references in multi-configuration and dynamic PHP-based web applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE CS, 2013.

[105] 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 *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[106] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. BabelRef: Detection and renaming tool for cross-language program entities in dynamic web applications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1391–1394, Piscataway, NJ, USA, 2012. IEEE Press.

[107] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. DRC: A detection tool for dangling references in PHP-based web applications. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1299–1302, Piscataway, NJ, USA, 2013. IEEE Press.

[108] F. Nielson and H. R. Nielson. *Two-level Functional Languages*. Cambridge University Press, 1992.

[109] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: An efficient and economical way of reducing slice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 422–431. ACM, 1999.

[110] N. I. of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/planning/upload/report02-3.pdf, 2002. Accessed: 2015-11-21.

[111] A. Orso, S. Sinha, and M. Harrold. Incremental slicing based on data-dependence types. In *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 158–167. IEEE CS, 2001.

[112] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[113] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, Apr. 1984.

[114] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125. Springer-Verlag, 2009.

[115] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical report, 1992.

[116] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE Computer Society, 2008.

[117] U. Praphamontripong and J. Offutt. Applying mutation testing to web applications. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 132–141, Washington, DC, USA, 2010. IEEE Computer Society.

[118] Quercus. Quercus website. http://quercus.caucho.com/, 2015. Accessed: 2015-11-21.

[119] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 129–138, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[120] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, pages 432–449. Springer-Verlag New York, Inc., 1997.

[121] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[122] F. Ricca and P. Tonella. Web application slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 148–157. IEEE Computer Society, 2001.

[123] F. Ricca and P. Tonella. Construction of the system dependence graph for web application slicing. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 123–132. IEEE Press, 2002.

[124] D. J. Richardson and L. A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 244–253, Piscataway, NJ, USA, 1981. IEEE Press.

[125] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of International Conference on Software Engineering*, pages 277–287. IEEE Press, 2012.

[126] R. Santelices, J. A. Jones, Y. Yanbing, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 56–66. IEEE Computer Society, 2009.

[127] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[128] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 422–432. ACM, 2013.

[129] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 488–498. ACM Press, 2013.

[130] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 263–272. ACM Press, 2005.

[131] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[132] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122. ACM, 2007.

[133] I. L. Stats. Internet live stats website. http://www.internetlivestats.com/, 2016. Accessed: 2016-02-18.

[134] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 203–217. ACM Press, 1997.

[135] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014.

[136] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM Press, 2012.

[137] Tidy. Tidy website. http://tidy.sourceforge.net/, 2015. Accessed: 2015-11-21.

[138] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[139] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[140] P. Tonella and F. Ricca. Web application slicing in presence of dynamic code generation. *Journal of Automated Software Engineering*, 12(2):259–288, 2005.

[141] W3C. CSS selectors. http://www.w3.org/TR/CSS21/selector.html, 2015. Accessed: 2015-11-21.

[142] WALA. WALA tools in JavaScript. http://wala.sourceforge.net/wiki/index.php/Main_Page#WALA_Tools_in_JavaScript, 2015. Accessed: 2015-11-21.

[143] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings in web applications. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 87–96. ACM Press, 2010.

[144] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 32–41. ACM, 2007.

[145] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[146] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 249–260, New York, NY, USA, 2008. ACM.

[147] M. Weiser. Program slicing. *IEEE Trans. Softw. Engineering*, 10(4):352–357, Oct. 1984.

[148] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng.*, 6(3):236–246, May 1980.

[149] Wikipedia. Client-server model. https://en.wikipedia.org/wiki/Client%E2%80%93server_model, 2016. Accessed: 2016-03-07.

[150] Wikipedia. Symbolic execution. https://en.wikipedia.org/wiki/Symbolic_execution, 2016. Accessed: 2016-03-07.

[151] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[152] J. Xu, Y. Gao, S. Christley, and G. Madey. A topological analysis of the open souce software development community. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, page 198.1. IEEE Computer Society, 2005.

[153] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 251–260, New York, NY, USA, 2011. ACM.

[154] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 272–281. ACM, 2006.

[155] Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao. Virtual dom coverage for effective testing of dynamic web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 60–70, New York, NY, USA, 2014. ACM.