Automated Identification of Parameter Mismatches in Web Applications

William G.J. Halfond and Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfond, orso}@cc.gatech.edu

ABSTRACT

Quality assurance techniques for web applications have become increasingly important as web applications have gained in popularity and become an essential part of our daily lives. To integrate content and data from multiple sources, the components of a web application communicate extensively among themselves. Unlike traditional program modules, the components communicate through interfaces and invocations that are not explicitly declared. Because of this, the communication between two components can fail due to a parameter mismatch between the interface invoked by a calling component and the interface provided by the called component. Parameter mismatches can cause serious errors in the web application and are difficult to identify using traditional testing and verification techniques. To address this problem, we propose a static-analysis based approach for identifying parameter mismatches. We also present an empirical evaluation of the approach, which we performed on a set of real web applications. The results of the evaluation are promising; our approach discovered 133 parameter mismatches in the subject applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Web applications

1. INTRODUCTION

Web applications provide a diverse range of services, from online banking to multi-player games, and have become an important part of the daily lives of millions of users. In fact, feature-rich and sophisticated web applications contribute heavily to the success of many well known companies, such as Amazon, eBay, and ING Direct. For this reason, quality assurance techniques targeted at web applications are increasingly important as companies work to ensure that their applications perform reliably and meet the expectations of their users.

The components of a web application communicate extensively in order to provide a feature-rich environment that integrates content and data from multiple sources. Communication between web components is different from that between traditional program modules. When a web component A communicates with another component B, it does so by sending a request to B that invokes one of B's interfaces and provides a set of arguments in the form of name-value pairs. An error in this communication can occur for several reasons: B may expect additional arguments, A or B may refer to the same argument by different names, or A may send too many arguments. We call these types of errors *parameter mismatches*.

For modern web applications, parameter mismatches have become a serious and common problem. In fact, a recent empirical study [7] reported that parameter mismatches are one of the most frequent types of errors made by web application developers. The complexity of inter-component communication, where both the generation of interface invocations and the definition of a component's accepted interfaces occur at runtime, contributes to the likelihood of these errors occurring. Since parameter mismatches affect the ability of web components to communicate correctly, the results of this type of error can be serious and can cause a component to fail unexpectedly or return incorrect results.

Automatically identifying parameter mismatches in modern web applications is challenging for current testing and analysis techniques. Many current techniques were designed for simple static web applications, which contain hard-coded interface invocations. For these applications, it was sufficient to inspect their HTML code and ensure that each invocation contained the correct arguments and correlated with an accepted interface of the target component. For modern web applications, which are generally more complex and dynamic, the presence of implicitly defined accepted interfaces and dynamically generated invocations precludes such a straightforward solution.

In this paper we propose a new approach that automatically identifies and reports parameter mismatches in web applications. Our approach can handle complex web applications that dynamically generate interface invocations in HTML and have implicitly-defined accepted interfaces. Our approach consists of three main steps: 1) determine the set of interfaces accepted by each web component, 2) determine the set of interfaces invoked by each web component, and 3)

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

check whether each interface invocation matches an accepted interface of the invocation's target. In previous work [12] we presented a technique for performing the first step of this approach. In this work, we propose a technique for performing the second and third steps of the approach. We also present the results of an evaluation that we performed to assess the effectiveness and practicality of our approach. For the evaluation we developed a prototype implementation of our approach and applied it to four real web applications. The results of our evaluation are positive; our approach was able to identify parameter mismatches in all of the subject web applications.

The contributions of this paper are:

- An approach for identifying and reporting parameter mismatches in web applications.
- An implementation of the approach in a prototype tool.
- Two empirical studies that assess effectiveness and practicality of our approach on a set of real web applications.

2. BACKGROUND AND EXAMPLE

In this section we provide background information on web applications and define terminology that we use in the rest of the paper. We also introduce an example web application to illustrate relevant concepts and serve as a running example in the explanation of our approach in Section 3.

A *web application* is a software system that is available to users over the Internet. Web applications consist of *web components*, which are the modules that implement the different parts of the application's functionality, such as logging in a user, displaying an order, or processing a request. Each component provides a *root method*, which is the entry point for the component and is the first method called when the component executes.

To invoke the functionality of a component, a client (*e.g.,* a web browser or another component) submits a request to the target component. The request contains zero or more arguments, which are represented in the form of name-value pairs. A web server receives the request on behalf of the target component, calls the root method of the component, and passes to the method a request object that contains the set of name-value pairs it received in the initial request. The web component performs its intended computation and produces an HTML page that is returned to the client.

Figure 1 provides a depiction of a workflow of an example web application. The web application is hosted on a server at `site.com`. It allows users to search a listing of businesses and restrict the returned results by zip code, type of business, or city and state. The application consists of two components, `searchpage.jsp` and `dosearch.jsp`. The components are implemented as Java *servlets*, which are classes that implement a specific `Servlet` interface. This interface specifies, among several things, that the class must implement a root method named `service`. Figures 2 and 3 show excerpts of the two servlets' code. Note that, although the example is written in Java, our approach is generally applicable for other web development languages, such as PHP, ASP, Perl, or any other web application that uses the Common Gateway Interface (CGI) calling convention. Also, to keep our example concise, we have omitted much of the complexity found in real web applications. Real web applications can contain tens of thousands of lines of code, multiple layers of abstraction, and use complex expressions to generate invocations and obtain parameter values.

In Figure 1(a) a user makes a request to `searchpage.jsp`, by requesting the URL `http://site.com/searchpage.jsp`. The server at `site.com` receives the request and calls method `service` of `searchpage.jsp`, passing as a parameter a request object with no name-value pairs because the request from the user did not contain any arguments. When `searchpage.jsp` receives the request, it first outputs the opening HTML tags (lines 1–2), a `<form>` tag (line 3), and an `<input>` tag (line 4). A `<form>` tag is rendered by the browser as a web form that allows users to submit data to a web application using input fields. Inputs can be specified using various tags, such as `<input>` and `<select>`. For example, the `<input>` tag produced at line 4, adds a text input box to the web form. Depending on the value of variable `conf.searchPref`, `searchpage.jsp` generates one of three web forms: 1) if the value is "zip," it adds an input field to let the user specify a zip code (lines 7–9); 2) if the value is "type," it calls method `printTypes` to add a drop-down list box for selecting one of several predefined business types (lines 24–31); 3) in all other cases, the servlet adds two input fields to the form that allow the user to specify a city and a state (lines 13–18). To each of the forms, the servlet then adds a hidden input field to keep track of the search preference and a submit button that is used to trigger the form submission (lines 20–21). Finally, the servlet outputs the closing HTML tags (lines 22–23). The resulting web form is then returned to the client and displayed to the user.

Figure 1(b) shows what happens when the user interacts with the web form based on the type of business. If the user specifies "bagels" as the search string, selects "Bakery" as the type of business, and presses the submit button, the client generates the following URL: "`http://site.com/dosearch.jsp?search=bagels&business=Bakery`." This URL specifies the address of the target servlet and two parameters encoded as name-value pairs. Note that the name of the target servlet is specified in the action field of the `<form>` tag (line 3 in Figure 2).

After processing the URL, the server invokes method `service` of `dosearch.jsp` and passes the parameters in as a request object. When invoked, `dosearch.jsp` initializes an SQL query string, retrieves the input parameter named "search," which contains the search string entered by the user, and creates the first part of a database query that will search for the name specified by the user (lines 1–3). Depending on the value of "searchPref" (retrieved at line 4), the servlet retrieves "zip" (lines 5–7), "busines" (lines 8–10), or "state" (lines 11–13) from the request object, and uses this information to refine the database query. Finally, `dosearch.jsp` executes the database query and generates a page with the results (lines 15–16).

Interaction between a client and a target component occurs via interfaces. Each web component provides one or more accepted interfaces and performs zero or more interface invocations. A component's *accepted interfaces* are interfaces that allow for invoking the component's functionality. We represent an accepted interface as follows:

*accepted interface=component, parameter**

Each accepted interface consists of a *component*, which is the name of the web component to which the interface belongs, and zero or more *parameter*s, uniquely identified by their name. Unlike traditional applications, where interfaces are explicitly defined (*e.g.,* the public methods in

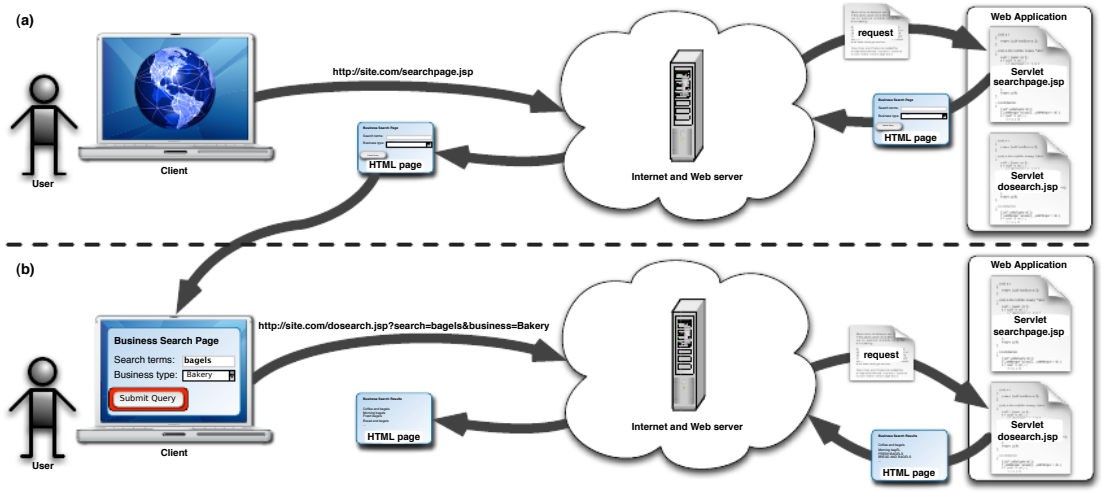

Figure 1: Sample work-flow for the example web application.

```
   ...
   void service(Request req) {
1.   print("<html><body>");
2.   print("<h1>Business Search Page</h1>");
3.   print("<form method=GET action=dosearch.jsp>");
4.   print("<input type=text name=search>");
5.   print("<br>");
6.   if (conf.searchPref.equals("zip")) {
7.     print("<font color=black>Zip:</font>");
8.     print("<input type=text name=zip>");
9.     print("<br>");
10.  } else if (conf.searchPref.equals("type")) {
11.    printTypes();
12.  } else {
13.    print("<font color=black>City:</font>");
14.    print("<input type=text name=city>");
15.    print("<br>");
16.    print("<font color=black>State:</font>");
17.    print("<input type=text name=state>");
18.    print("<br>");
19.  }
20.  print("<input type=hidden name=searchPref value="
           + conf.searchPref + ">");
21.  print("<input name=Submit type=submit>");
22.  print("</form>");
23.  print("</body></html>");
   }

24. void printTypes() {
25.   print("<select name=business>");
26.   optValues[] = ["Jewelry", "Bakery", "Restaurant"];
27.   for (String opt:optValues) {
28.     print("<option value=" + opt + ">" + opt + "</option>");
29.   }
30.   print("</select><br>");
31. }
   ...
```

Figure 2: Code excerpt for servlet searchpage.jsp.

a Java library class), accepted interfaces for a web application are implicitly defined by the set of parameters accessed along different execution paths. For all but the simplest components, accurately determining the accepted interfaces requires an analysis of the web components' code. In previous work, we developed an approach for identifying accepted interfaces based on static analysis [12]. The results of running this analysis on `dosearch.jsp` are shown in Table 1. The three interfaces correspond to the three possible paths through the servlet. Each interface contains the parameters named "search" and "searchPref," as these parameters are accessed along every path of execution. The first interface corresponds to the path that includes lines 5–7 and contains the parameter named "zip." The second interface

```
   ...
   void service(Request req) {
1. String dbQuery = "select businesses from db where ";
2. String search = req.getParameter("search");
3. String dbQuery += "name like '" + search + "' ";
4. String searchType = req.getParameter("searchPref");
5. if (searchType.equals("zip")) {
6.    String zip = req.getParameter("zip");
7.    dbQuery+="zip="+zip;
8. } else if (searchType.equals("type")) {
9.    String type = req.getParameter("busines");
10.   dbQuery+="type="+type;
11. } else {
12.   String state = req.getParameter("state");
13.   dbQuery+="state="+state;
14. }
15. ResultSet results = execute(dbQuery);
16. print(results);
   ...
```

Figure 3: Code excerpt for servlet dosearch.jsp.

Table 1: Accepted interfaces servlet `dosearch.jsp`.

	Accepted Interface 1	Accepted Interface 2	Accepted Interface 3
Component:	dosearch.jsp	dosearch.jsp	dosearch.jsp
Parameters:	search searchPref zip	search searchPref busines	search searchPref state

corresponds to the path that includes lines 8–10 and contains the parameter named "busines." The third interface corresponds to the paths that includes lines 11–13 and contains the parameter named "state."

We refer to the requests made by a component to another component as *interface invocations*. Similar to accepted interfaces, we represent an interface invocation as follows:

*interface invocation=target, argument**

Each invocation consists of a *target*, which is the name of the invoked web component, and a set of *argument* names. A component can perform either a direct or indirect interface invocation. To perform a *direct* interface invocation, a component encodes the invocation as a URL and passes it to an API method that performs the invocation and returns the result to the client. To perform an *indirect* interface invocation, a web component generates an HTML page that contains interface invocations and sends the page to a web browser. The page contains specific HTML tags that enable the browser to make an invocation

on behalf of the component. An example of this is when one of the web forms in `searchpage.jsp` is submitted, it generates the URL "`http://site.com/dosearch.jsp?search=bagels&business=Bakery`." This sends an interface invocation to `dosearch.jsp`, and indirectly allows `searchpage.jsp` to invoke `dosearch.jsp`'s functionality. An indirect invocation can also occur via an `<a>` or `<img>` tag; the tags can contain a URL either as the link target or image source and cause the browser to make a specific interface invocation when the link is clicked or the image loaded. Similar to accepted interfaces, interface invocations are defined at runtime, and a single web component can generate many different invocations. In fact, each path of execution through a component can generate a unique set of interface invocations.

Parameter mismatches can cause a web application to fail at runtime. Our example web application has two such parameter mismatches that cause the application to exit abnormally or return incorrect results. The first mismatch occurs because of the misspelling of the name of the parameter used to specify the type of business to be searched. In `searchpage.jsp`, this parameter is named "business," whereas in `dosearch.jsp`, the parameter is referred to as "busines," omitting the last "s." This error causes a parameter mismatch that results in a null pointer exception when the servlet tries to dereference variable `type`. The second mismatch occurs because `searchpage.jsp` allows users to restrict search criteria by city and state, but `dosearch.jsp` never retrieves the value of parameter "city" provided by the search request. The result of this mismatch is that the user receives results that have not been correctly filtered based on the specified city and state. In our simple example, these errors could probably be found via manual inspection. In real web applications, however, finding these types of problem is extremely challenging due to the applications' complexity.

Errors related to interface invocations can be difficult to detect through testing for several reasons. First, testing is inherently incomplete, so it cannot guarantee that a component will generate all possible interface invocations. Second, identifying when an error related to a parameter mismatch has occurred is difficult. In many cases, a parameter mismatch would not result in an easily-observable failure, such as an exception. For instance, the second error in our example would be difficult to detect as it will only cause a deviation from correct behavior in cases where filtering based on the city field is relevant to the results. Our approach is not affected by these limitations. Since our analysis is conservative, it examines all possible interface invocations. Additionally, it can identify erroneous invocations accurately since it also knows the accepted interfaces of the target of the invocations.

To date, compilers are unable to detect parameter mismatch errors in web applications. The reason for this is that invocations are created in a generated object-language, such as HTML, and the accepted interfaces they target are only implicitly defined. The role of compilers is currently limited to verifying the syntactical correctness of the source code of the web application that generates the invocation, but not the correctness of the invocation itself. For example, a compiler can check that the parameter passed to an API call that performs a direct invocation is a string, but cannot determine that that name-value pairs specified in the string representation of the invocation correspond to an accepted interface of the target web component. Similarly, HTML validators could check to ensure that a generated HTML page is syntactically correct, but cannot verify that all indirect invocations it contains correspond to an accepted interface of the target components.

3. OUR APPROACH

The goal of our approach is to automatically identify parameter mismatches in web applications. To achieve this goal, our approach performs a static analysis of the web application. The approach consists of three main steps. In Step 1, the approach identifies the accepted interfaces of each component in the web application. In Step 2, the approach analyzes each component to determine its set of interface invocations. In Step 3, the approach checks whether each interface invocation matches an accepted interface of the invocation's target and reports any mismatches. In the following sections, we present each step of our approach in more detail and explain how these issues are handled by our analysis.

3.1 Step 1: Identify Accepted Interfaces

The goal of the first step of the technique is to identify the accepted interfaces of each web component. To do this, we leverage a technique that we developed in previous work [12] and that is based on a two-phase static analysis approach. In the first phase, the analysis computes domain information (*i.e.,* types and possible values) for each parameter accessed in the component. In the second phase, the analysis identifies the parameter names and groups them into logical interfaces based on the component's control and data flow. (Note that the technique also computes parameter domain information, but we do not utilize this information in the approach presented in this paper.) Table 1, which we discussed in Section 2, shows the accepted interfaces of `dosearch.jsp` (see Figure 3) that would be computed during Step 1.

3.2 Step 2: Determine Interface Invocations

The second step of our technique identifies each web component's set of direct and indirect interface invocations. As discussed in Section 2, components perform direct interface invocations by creating a URL containing the invocation and passing it to a specific API method. To identify such invocations, our technique visits each node of the component's inter-procedural control flow graph (ICFG), identifies all call sites that invoke API methods used to make direct invocations, and analyzes the parameters of these calls to extract the invocation URL. As an example, in Java the parameter containing the URL is represented as a string. Our technique determines the value of the URL using string analysis (see Section 3.2.1) and then parses the URL to identify information about the invocation's target and arguments. This information is later used in Step 3 of the analysis.

Identifying indirect invocations is considerably more complex than identifying direct invocations. Intuitively, our technique computes a conservative approximation of the set of HTML pages that can be generated by a component and then analyzes these pages to identify indirect invocations. To make the approach practical, our technique relies on a worklist-based data-flow analysis and a modular analysis based on the use of method summaries. The basic idea is to analyze each method or group of strongly connected methods of the component in reverse topological order with respect to the call graph and compute a summary of the

HTML fragments generated by the method. The use of method summaries allows our technique to analyze methods only once and reuse the analysis results for a method m at m's callsites. Reverse topological ordering is used to ensure that a method's summary is available before it is called. Iterative data-flow analysis is used within each method (or set of methods mutually involved in recursion) to compute summary information. To reduce the size of the summaries, HTML fragments are minimized by removing HTML content unrelated to the definition of interface elements. At the end of the analysis, the summary for the root method represents a conservative approximation of the pages that can be generated by the component. This set of pages is then analyzed by an HTML parser to extract the elements that represent interface invocations.

In the following sections, we discuss the technique for identifying indirect invocations in detail. In Section 3.2.1, we define supporting analysis and functions used by our technique. The algorithms that are at the core of the technique are presented in Sections 3.2.2, 3.2.3, and 3.2.4. Section 3.2.5 illustrates the algorithms using our running example.

3.2.1 Supporting Functions and Analyses

Our technique uses a *string analysis* based on the Java string analysis (JSA) by Christensen, Møller, and Schwartzbach [5]. The string analysis takes as input a reference to a string variable at a given point in an application and computes a conservative approximation of the values the string variable can assume at that point. The analysis is performed by analyzing the control and data flow of the application and modeling the string manipulation operations performed on the string variable. Our string analysis is based on JSA, but is limited in scope to the method that contains the reference to the string variable. If a string variable is partially defined by one of the method's parameters or by a global variable, we put a placeholder in the computed string value. A *placeholder* is a marker that specifies which of the method's formal parameters should be used to complete the string value when the method is called at a specific callsite. The use of placeholders allows us to make the string analysis context-sensitive because at each callsite we can substitute the value of the unknown parameter and more accurately calculate the possible values of the string variable.

In the description of our algorithms, we assume the availability of several standard helper functions that operate on a node n of a method's control flow graph (CFG): $target(n)$ returns the methods called at a call site n; $succ(n)$ returns all successors of n in n's CFG; and $pred(n)$ returns all predecessors of n in n's CFG. Additionally, we assume that the CFG for a given method is globally available.

3.2.2 Algorithms

Algorithms 1 and 2, `ExtractPages` and `SummarizeMethod`, extract a component's indirect interface invocations. The extraction starts in algorithm `ExtractPages`, which takes as input the call graph for the web component to be analyzed, CG. The algorithm first identifies the sets of strongly connected components in CG, SCC. These components represent sets of methods that call each other recursively and are treated as one "super-method" and assigned the same summary. The algorithm also initializes another set, CC, to contain the method sets in SCC and a singleton set for each node of CG that is not in SCC. Therefore, at the end of this step, every node of CG is in CC, either as a member of a method set from SCC or as a singleton set. Next, `ExtractPages` iterates in reverse topological order over each method set, $mset$, of CC and calls `SummarizeMethod` to create a summary that is associated with each method in $mset$. Lastly, `ExtractPages` passes the summary of the root method to an HTML parser to identify its indirect invocations.

Algorithm `SummarizeMethod` creates a summary for each method in $methodset$. Each *method summary* created by the algorithm is comprised of a set of strings that represent the distinct page fragments generated by the method. A *page fragment* represents HTML content that could be generated by an invocation of the associated method, and is comprised of HTML tags and placeholders for content that cannot be resolved within the method.

The core of algorithm `SummarizeMethod` is a worklist-based iterative data-flow analysis that summarizes information along all paths without having to analyze each path individually (which would be computationally infeasible for most non-trivial programs). The running time of the algorithm, when nodes are processed in an optimal ordering, is proportional to the number of nested loops in the code. The algorithm is guaranteed to converge because the transfer function that updates each ordered set in the Out set (lines 20–22) causes the sets to monotonically grow, and the sets' size is limited by the total number of nodes in the method's CFG.

First, `SummarizeMethod` initializes the algorithm's data structures. Set N is initialized with all of the nodes of the methods in $methodset$ (line 1). For each node n in N, the algorithm initializes Gen[n] in one of several ways, depending on the contribution n makes to the HTML page generated by the component. If n is a method entry point, Gen[n] is initialized to contain the empty set, and n's successor nodes are added to the worklist for later processing (lines 4–7). If n contains a call to either a function that writes to the component's output stream or a function with a summary, Gen[n] is initialized with n itself (lines 8–11). Lastly, if none of the previous conditions hold, Gen[n] is empty (line 13).

After initializing the Gen set, `SummarizeMethod` processes each node n in $worklist$ (lines 16–31). The processing begins by calculating In[n] as the union of the Out sets of n's predecessors (line 18). `SummarizeMethod` then computes Out[n] by appending the contents of Gen[n] to each set in In[n] (lines 19–22) and compares the new value of Out[n] against its old value (line 23). If no change has occurred, the processing of n is done, and the next node in $worklist$ is processed. If the value has changed, Out[n] is updated (line 24), and the successors of n are added to the worklist. If n is a callsite, and its target is one of the other methods in $methodset$, then the entry node of the target is added to $worklist$ (lines 25–26). Otherwise, the nodes returned by $succ(n)$ are added to $worklist$ (line 27). The processing of the nodes continues in this manner until $worklist$ is empty.

After processing the worklist, `SummarizeMethod` translates the ordered sets of nodes into page fragments. For each method m in $methodset$, `SummarizeMethod` iterates over the Out set associated with the exit node of m (lines 32–44). Each element of Out is an ordered set, $nodeset$, which contains nodes that generate HTML along a single control flow path of the method. For each node n in $nodeset$, the algorithm does the following: 1) call `resolve` to perform HTML fragment resolution, (see Section 3.2.3) which determines

The analysis treats globals as additional parameters.

Alg. 1 – Identify Interface Invoc. – ExtractPages

1:
2:
3:
4:
5:
6:
7:
8:
9:

the HTML fragments contributed by the node (line 37), 2) append the node's HTML content to the HTML content generated by the previous nodes in *nodeset* (line 38), and 3) call function `reduce` (line 39) to perform intermediate parsing on the HTML fragments, which removes HTML tags that do not contribute to the definition of interface elements (see Section 3.2.4). After every *nodeset* has been analyzed, *methodsummary* contains the HTML fragments to be associated with method m (line 43).

3.2.3 HTML Fragment Resolution

The process of HTML fragment resolution determines the HTML content contributed by a given node of a web component. In our approach, HTML fragment resolution is implemented by function `resolve`, which is called at line 37 of algorithm `SummarizeMethod`. Function `resolve` takes a node n as input and returns a set of strings (possibly with placeholders) that represent the HTML fragments contributed by the node.

The resolution of n proceeds in two ways depending on whether n writes data to the component's output stream or calls a method that is associated with a summary. In the first case, if n writes data to the output stream, `resolve` runs the string analysis on the argument that contains the data to be written. Function `resolve` returns the set of strings computed by the string analysis as n's contributed HTML content. In the second case, if n calls a method with a summary, `resolve` retrieves the summary associated with the target method of n. If the summary contains any placeholders, `resolve` runs the string analysis on the corresponding arguments provided by the callsite and replaces the placeholders with the results of the string analysis. Function `resolve` then returns the substituted strings as n's contributed HTML content.

In some cases, a placeholder cannot be resolved even after processing the root method of a component. This happens when the placeholder represents external input to the component, such as user input or data read from a file. In this scenario, our approach assumes that it can ignore the placeholder, but generates a warning to notify developers of the situation. This assumption is unsafe only in cases where external input could contain HTML that affects the definition of interface-related tags; in that case, our assumption could lead to either false positives or false negatives, depending on the intended content of the external fragments. However, this assumption is rarely violated and did not occur for any subject we analyzed. Moreover, if needed, it would be straightforward to incorporate a mechanism that lets developers specify the possible content of external fragments.

Alg. 2 – Identify Interface Invoc. – SummarizeMethod

1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28:
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:

3.2.4 Intermediate Parsing

Intermediate parsing reduces the amount of string data that must be stored and propagated by the analysis. In our approach, intermediate parsing is implemented by function `reduce`, which is called at line 39 of algorithm `SummarizeMethod`. Function `reduce` takes an HTML fragment as input and returns an HTML fragment in which irrelevant tags have been removed.

The motivation for using intermediate parsing is that storing all HTML fragments that can be generated by a component creates a high memory overhead for the analysis. One insight that allows us to reduce the overhead is that many of the HTML fragments contain tags that do not affect interface invocations and are only used to display text or visually enhance a web page. Examples of these tags include `<font>`, `<hr>`, and `<br>`. Such tags can be removed from the propagated strings without affecting the analysis results.

To perform intermediate parsing, algorithm `reduce` uses a customized HTML parser that tokenizes the input strings into individual HTML elements. Algorithm `reduce` removes all tags that do not define an interface. In order to be safe and not affect the accuracy of the interface extraction, `reduce` only removes tags that can be completely identified

in the parsed string and that do not involve the use of any placeholders introduced by the `resolve` function. These two conditions are necessary in order to avoid removing tags that are either only partially completed because their construction spans several nodes or whose final structure may vary once a placeholder has been resolved.

In our experience with intermediate parsing, we noticed that, in addition to the size reduction of each string, eliminating tags also helps to expose duplicate interface-related fragments. This happens because many of the propagated strings vary only in the substrings that define tags unrelated to interface invocations. When these tags are eliminated, the strings contain the same interface-related tags, and the duplicate entries can be eliminated. Since line 38 of `SummarizeMethod` computes the Cartesian product of the propagated strings, this results in significant savings. Case in point, the analysis of one large web component without intermediate parsing produced almost 23 million page variations. By employing intermediate parsing, this number was reduced to less than 4,500.

3.2.5 Illustration with Example Servlet

We illustrate the process of extracting interface invocations using the example servlet `searchpage.jsp` from Figure 2. We assume that the algorithms operate on a CFG with a one to one mapping of nodes to lines of code.

The summarization begins with method `printTypes`. Because lines 25, 28, and 30 of the servlet write to the component's output stream, for each $l \in \{25, 28, 30\}$, Gen[l] is initialized to $\{l\}$. The algorithm updates the data-flow equations for each line. Because the control flow proceeds from line 25 to 30, the Out set of `printTypes`' exit node is {{25, 28, 30}}. After generating this solution for the data-flow equations, the algorithm calls `resolve` and `reduce` on the Out set of the method's exit node. Line 25 resolves to the string value of the `<select>` tag it prints. The resolution of line 28 returns a string containing three `<option>` tags, one for each of the values in the array at line 26, and line 30 resolves to the closing `</select>` tag. The HTML fragment comprised of the opening `<select>` tag, followed by the three `<option>` tags, and closed by the `</select>` tag is associated with `printTypes` as the method summary.

Processing of the root method begins by initializing the Gen set of each line of the servlet. The Gen set of the special entry node of the method contains the empty set, and each line $l \in \{1-5, 7-9, 13-18, 20-23\}$ is initialized so that Gen[l] $\leftarrow \{l\}$. Gen[11] is initialized to {11} because it calls `printTypes`, which has a summary. The worklist iteration begins with line 1 of the servlet. In[1] is the union of the Out sets of line 1's predecessors. The only predecessor to line 1 is the entry node, whose Gen set contains the empty set; therefore, Out[1] $\leftarrow$ {{1}}, which is the result of appending Gen[1] to each element in its In set. Processing continues through line 5, where the Out set is {{1,2,3,4,5}}. At this point, control flow can continue at either lines 6, 10, or 12. In[6] is equal to Out[5], and processing continues until line 9, where the Out set is {{1,2,3,4,5,7,8,9}}. Control flow also continues at line 10, where the In set is Out[5]. Processing Line 11 adds node 11 to its Out set, which becomes {{1,2,3,4,5,11}}. Lastly, for the control flow continuing at line 12, In[12] is also equal to Out[5]. Processing continues until line 18, where the Out set is {{1,2,3,4,5,13,14,15,16,17,18}}. All three control flow paths are joined at line 20, so In[20] is the union of the Out sets of lines 9, 11, and 18, {{1,2,3,4,5,7,8,9}, {1,2,3,4,5,11}, {1,2,3,4,5,13,14,15,16,17,18}}. Processing lines 20–22 adds references to these nodes to each of the three sets. The final result is that the Out set of the exit node of the method is: {{1,2,3,4,5,7,8,9,20,21,22}, {1,2,3,4,5,11,20,21,22}, {1,2,3,4, 5,13,14,15,16,17,18,20,21,22}}.

Table 2: Interface invocations for servlet `searchpage.jsp`, shown in Figure 2.

	Interface Invocation 1	Interface Invocation 2	Interface Invocation 3
Target:	dosearch.jsp	dosearch.jsp	dosearch.jsp
Arguments:	search searchPref zip	search searchPref business	search searchPref city state

After calculating the Out set of the exit node, `SummarizeMethod` determines the HTML fragments generated by each of the three sets in the exit node's Out set. For the purpose of this example, we will examine the processing of the second set, {1,2,3,4,5,11,20,21,22}, which corresponds to the path through the *true* branch at line 10. Algorithm `SummarizeMethod` begins to iterate over each line in the set. It first calls the `resolve` function on line 1. The generated string for this line is "`<html><body>`", which becomes the initial value of the method's HTML fragment. Because both of these tags are compound tags (*i.e.,* they have an open and close tag), neither can be safely removed yet by the `reduce` function, even though they do not contribute to the definition of any interface elements. Processing of line 2 produces string "`<h1>Business Search Page</h1>`," which is appended to the current HTML fragment. The subsequent call to `reduce` eliminates the `<h1>` tag and its content because it does not contribute to the definition of any interface elements. Therefore, after the reduction, the HTML fragment has the same value as it did after the first iteration. Processing continues similarly for lines 3, 4, and 5, with only the results of line 5 being discarded because `<br>` does not define interface elements. Calling `resolve` on line 11 returns the summary of `printTypes`. Processing of line 20 contributes an `<input>` tag. Since the value of variable "searchPref" cannot be resolved, a placeholder is returned for the portion of the input tag it defines. Lines 21 and 22 also contribute to the generation of the HTML fragment. Line 23 closes the `<html>` and `<body>` tags, which means that the `reduce` function can now safely eliminate the opening and closing tags from the HTML fragment. At this point, the summary for the root method is complete and consists of the HTML generated by lines 3, 4, 25–29 (in lieu of the function call at line 11), and 20–22. The placeholder is ignored since it cannot be resolved in the root method. The HTML fragment for the root method is then passed to an HTML parser, which identifies the different interface elements and returns the interface invocation shown as "Interface Invocation 2" in Table 2. Note that input elements that are of type submit are ignored by the analysis since they do not contain user input. The interface invocations corresponding to the first and third set in the Out set of the root method's exit node are listed in Table 2 as "Interface Invocation 1" and "Interface Invocation 3," respectively.

Alg. 3 – Verify Interface Invocations – VerifyInterfaces

1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:

3.3 Step 3: Verify Interfaces

The third and final step of our approach is to verify each component's interface invocations. For each identified invocation, our approach identifies the target of the invocation and checks that the invocation matches one of the target's accepted interfaces. An invocation *matches* an accepted interface of its target if the invocation's set of argument names equals the names of the parameters in the accepted interface. Each invocation that does not match an accepted interface is reported to the developers as a potential error.

Algorithm 3, `VerifyInterfaces`, takes as input a component's set of interface invocations and reports the unmatched invocations. For each invocation, *invk*, the algorithm first identifies the target component of the invocation (line 3) and the accepted interfaces of the target component (line 4). Before beginning the main loop, the algorithm initializes *invokeok* to false (line 5). This boolean flag is used to track when a matching interface has been found in the accepted interfaces. The algorithm then iterates over each accepted interface, *interface* (lines 6–10). `match` returns true only if *invk* matches *interface* (line 7). When a match is found, the *invokeok* flag is set to true (lines 8). Finally, if there is no match, *invk* is added to *mismatches* (lines 11–13) and *mismatches* is returned as the output of `VerifyInterfaces`.

3.3.1 Illustration with Example Servlet

To illustrate the verification step, we apply it to the interface invocations of the example servlet `searchpage.jsp`, which are shown in Table 2. We assume that the loop iterates over the invocations in numerical order. For the first invocation, the algorithm extracts the target name, `dosearch.jsp`, retrieves the target's accepted interfaces, and compares the invocation against each of the accepted interfaces. (The accepted interfaces for `dosearch.jsp` are shown in Table 1.) In this case, the first invocation matches the first accepted interface, so the invocation is not added to the list of mismatches. The second invocation does not match any of the accepted interfaces, due to the misspelling in the parameter denoting the type of business to be searched. It is therefore added to the list of mismatches. Similarly, the third invocation does not match any of the accepted interfaces because it contains an extra parameter, "city," which is not present in any of the accepted interfaces. The final list of mismatches returned by `VerifyInterfaces` consists of the second and third invocations, which correspond to the two errors discussed in Section 2.

3.4 Limitations of the Analysis

Several factors limit the accuracy of our technique. First, our technique, like most program analysis techniques, is affected by the presence of infeasible paths. The string analysis and iterative data flow analysis both assume that all paths are feasible in a given application. As a result, our technique computes a conservative over-approximation of the set of HTML pages that could be generated by a component and identifies interface invocations that could not occur at runtime. However, two factors limit the effect of infeasibility on the analysis results: 1) the use of placeholders in our analysis, during HTML fragment resolution, and 2) the typical developer practice of using hard-coded strings in statements that output HTML, which makes the string values easy to determine. In our empirical evaluation, infeasibility had no negative effect on our results.

The use of JavaScript to modify the creation of interface related elements in an HTML page can also affect the accuracy of our approach. For example, it is possible for JavaScript to add (remove) a name-value pair to (from) an invocation before it is performed. We could address this issue by extending our technique so that it analyzes and accounts for the effects of JavaScript functions. However, implementing this solution would require a considerable amount of engineering, so we did not implement it in our current prototype. Instead, we report all mismatches that arise due to this limitation as false positives.

Another factor that can affect the accuracy of our approach is the generation of HTML pages whose validity is dependent on looping structures that execute a specific number of times. Accurately handling this case would require an expensive path-sensitive analysis and preclude using the more efficient approach based on iterative data flow analysis. It is worth noting that, in all of the applications that we examined, we did not encounter any HTML generation that was dependent on loops executing a specific number of times. When loops were used to generate HTML, they typically looked like the one in lines 27–29 of Figure 2, which generates HTML tags for each element in an array. For these cases, the string analysis is able to determine the possible values of the strings generated by the loop, and our approach computes accurate results.

4. EVALUATION

The goal of the evaluation is to assess the usefulness and effectiveness of our approach. To perform this assessment, we measured the time required to run our analysis and its accuracy in identifying parameter mismatches. Our evaluation addresses the following two research questions:

RQ1: How efficient is our analysis when run on real web applications?
RQ2: What percentage of the reported parameter mismatches represent actual errors in the web applications?

In the following sections, we describe the implementation of the verification approach, the test subjects, and the studies we performed to address the research questions.

4.1 Tool Implementation

For the purpose of the evaluation, we developed a prototype implementation of our approach. The prototype, called Web Application Interface Verification Engine ( ), is written in Java and can analyze web applications built using

Table 3: Subject programs for our empirical studies.

Subject	*Description*	*LOC*	*Cls.*	*Svlt.*
Bookstore	Online bookstore.	18,919	28	25
Daffodil	Customer mgmt.	19,305	119	70
Filelister	Online file browser.	8,630	41	10
JWMA	Webmail.	29,402	97	20

Table 4: Study 1 – Timing results (s).

Subject	*Step 1*	*Step 2*	*Step 3*	*Total*
Bookstore	447	205	1	653
Daffodil	16,747	205	1	16,953
Filelister	1,712	66	1	1,779
JWMA	9,645	630	1	10,276

Java Enterprise Edition (JEE). Although the implementation targets only Java-based web applications, the approach is generally applicable to a wide range of other web development languages, such as PHP, ASP, and Perl. analyzes the classes in a web application and outputs a list of interface invocations that do not match any accepted interface. consists of three modules that implement the three phases of our approach. The first module, which extracts accepted interfaces, relies on our previously-developed tool [12]. The second module implements the algorithms described in Section 3.2. This module uses the Soot program analysis framework (`http://www.sable.mcgill.ca/soot/`) to generate call graphs and control-flow graphs, a modified version of JSA [5] to perform string analysis, and a customized version of HTML Parser (`http://htmlparser.sourceforge.net/`) to parse HTML pages and fragments. The third module implements the algorithm from Section 3.3.

4.2 Experiment Subjects

In the evaluation, we examined four Java web applications. Two of the applications—Daffodil (`http://sourceforge.net/projects/daffodilcrm/`, 20,253 downloads) and JWMA (`http://sourceforge.net/projects/jwma/`, 70,444 downloads)—are open source applications available from Sourceforge. The other two applications, Bookstore and Filelister, have been used in previous related work [10, 11]. For each application, Table 3 provides a brief description of the application (*Description*), its size (*LOC*), its number of classes (*Cls.*), and the number of these classes that are servlets (*Svlt.*).

4.3 RQ1: Efficiency

To address RQ1, we ran against our four experiment subjects and, for each subject, measured the time necessary to complete the three steps of the approach. We also noted cases in which it was not possible to run the analysis, due to either memory limitations or problems with libraries used in the implementation. We performed the experiments on a single machine with a Pentium D 3.0Ghz processor running GNU/Linux 2.6 and 2GB of memory, of which 1.5GB was dedicated to the heap space of the Java virtual machine (JVM). Table 4 shows the measurements for each of the subject applications. For each application, we show the time taken to perform each step (*Step 1*, *Step 2*, and *Step 3*), and the total amount of time for all three steps (*Total*).

The measurements of Table 4 show that the verification process for each application took from 1,779 to 16,953 seconds (*i.e.,* from 30 minutes to five hours, roughly). The analysis of Daffodil took the longest, as there was one class that needed almost three and a half hours to be analyzed. The analysis time for this class represented almost 87% of the total time spent analyzing Daffodil. We manually inspected the class and found that it had a complex control flow with many nested loops and branch statements. Most significantly, almost all of the statements in the class generated an interface-related tag. This meant that our analysis had to track a high number of distinct page fragments that related to interface invocations and was not able to incur any significant savings through intermediate parsing. For all of the subject applications, the most time consuming part of the analysis was processing done by external libraries, such as Soot building call graphs and JSA building flow graphs for the string analysis. In more fine-grained timing measurements, which we do not show here for space considerations, an average of 50% to 86% percent of the processing time for each application was spent within these libraries.

There were several instances in which was not able to analyze a particular class. This happened for two reasons: (1) the external libraries threw an error while analyzing the servlet, which happened for 24 servlets in the Daffodil subject, and (2) the analysis ran out of memory, which occurred for two servlets, one in JWMA and one in Filelister. We did not count the processing time of these classes in the totals of Table 4. For the classes that ran out of memory, we found that this occurred during HTML fragment resolution. The resolution created a high number of large, distinct HTML fragments that exceeded the heap space allocated to the JVM. A more efficient mechanism for storing HTML fragments (*e.g.,* a disk-based cache), would likely enable to handle these problematic classes.

The timing measurements obtained in our study indicate that our analysis, although expensive, is efficient enough to be incorporated into existing quality assurance processes. Our anecdotical evidence indicates that the execution time of our approach is significantly faster than manual inspection. Although we did not formally measure the time associated with manual inspection, our experience during the testing and evaluation of gives us a point of comparison. While developing , we verified the implementation by manually calculating sets of interface invocations and accepted interfaces for a subset of the servlets. Although we were familiar with the applications, it took close to 12 hours to inspect the code and derive the correct sets for just four classes.

It is also worth noting that is an unoptimized prototype implementation. For instance, we did not explore caching and reusing data structures built by Soot and JSA. As our fine-grained timing measurements showed, creating these structures for each class was a significant percentage of the execution time, and any reduction to this time would result in a considerable performance improvement for our approach. Most importantly, the envisioned typical usage scenario of our approach is to perform the analysis once per application (*e.g.,* overnight), and report the results to developers. Therefore, even the worst case of the analysis running for a few hours would not impede the practicality of our approach.

4.4 RQ2: Precision

To investigate RQ2, we ran on the subject applications and checked the parameter mismatches reported by the

Table 5: Analysis of invocation mismatches.

Subject	*# Acc.*	*# Invk.*	Mismatches					
			False Pos.			*Errors*		
			W	*J*	*R*	*O*	*S*	*I*
Bookstore	154	26	0	0	0	6	2	4
Daffodil	64	23	2	3	6	0	0	1
Filelister	25	4	0	0	0	1	2	0
JWMA	46	124	0	7	0	33	17	67

tool. We manually inspected the code that generated every mismatch to determine whether the mismatch represented an actual error or was a false positive. The inspection also enabled us to determine the root cause of the mismatch.

Table 5 shows the results of the second study. For each application, we list the number of accepted interfaces (*# Acc.*), the number of interface invocations (*# Invk.*), and the classification of the reported mismatches as either false positives (*False Pos.*) or actual errors (*Errors*). Based on our code inspection, we further classified the errors and false positives by root cause. These are listed in the table by the first letter of the root cause (*W, J, R, O, S, I*), which we define and discuss below.

As the results show, found 133 parameter mismatches that represented errors in the subject web applications. We classified each mismatch according to the following three root causes:

Ignored parameter: An argument in the invocation is not accessed by the target component.

Optional parameter: An invocation does not contain an argument that is accessed by the accepted interfaces of the target component. However, there is code that handles the case of the argument not being provided. Without a specification, it is not certain whether this is a misuse scenario that is gracefully handled by the developer or the parameter is intended to be optional. The technique reports these mismatches anyway because they represent potential errors.

Syntax error: An invocation does not match because of a misspelling in an argument name or a formatting error in the invocation (*e.g.,* in an HTML tag or URL string).

In our manual inspection of the code, we found that the impact of the errors varied widely. For example, in Filelister the two mismatches led to incorrect filtering of a search query and caused the application to return erroneous results to the end user. In Bookstore, three of the mismatches allowed a user to click a link to display updated information, but the intended action was not completed and an error message was not displayed to the user. The result of one of the mismatches in JWMA was that a data field associated with a customer's profile was not saved. Through our code inspection, we also found that the actual errors that led to the mismatches ranged from complicated logic to typos in the names of arguments and parameters. For example, four of the mismatches in Bookstore were due to erroneous logic in the target components that did not anticipate legal combinations of arguments. Conversely, the errors in Filelister and JWMA were caused by a syntax error in an indirect invocation and a misspelling of the name of an accessed parameter.

For two of the subject applications, did not generate any false positives. In the two applications that did have false positives, we identified three causes for the generation of the false positives:

WAM: These false positives are due to limitations of our tool ; may miss interfaces elements of the target component in cases where a web application uses non-standard ways of extracting parameters from a request object [12].

JavaScript: JavaScript code in a generated HTML page can add additional arguments to an invocation before it is submitted to the target component. Our analysis does not analyze JavaScript, so it cannot detect the change to the affected invocation.

Redirects and Imports: A web application component can redirect requests to other components or import code fragments that change the component's set of interface invocations or accepted interfaces. Our analysis currently does not account for the effects of redirections and imports.

As the results in Table 5 show, the two dominant root causes of false positives are "JavaScript" and "Redirects and Imports." Although addressing these limitations is conceptually straightforward, and would likely eliminate most of the related false positives, it would require non-trivial extensions to the technique. We therefore decided to postpone these extensions to a later stage of the research. The third root cause, "WAM," can be addressed by improving our analysis technique for accepted interfaces. We are currently working on an approach that uses symbolic execution to improve the accuracy of the analysis. Our preliminary results indicate that WAM-related false positives can be substantially reduced.

For all but one of the subjects, the results of the evaluation are positive. Out of 151 mismatches detected by our approach, only 18 are false positives. However, for Daffodil, our analysis reports one real error and eleven false positives. Although the ratio of errors to false positives is low in this case, it is worth noting that extensions to the technique could eliminate many, if not all, of the false positives. Overall, we find that the results are encouraging and suggest that our approach is an effective and useful technique for discovering parameter mismatches in web applications.

5. RELATED WORK

Related work includes several approaches that build models of web applications. An early technique by Ricca and Tonella [16] allows developers to use UML to model the links and interface elements of each page in a web application. Work by Betin-Can and Bultan [2, 4] provides a more expressive modeling languages that allows developers to represent dynamic interactions between components in a web application. These models can allow developers to verify the correct usage of invoked and accepted interfaces. However, the techniques are not fully automated because they rely on developer-provided specifications. Recent work performs a static analysis of the application and automatically build these models. Deng, Frankl, and Wang [6] model a web application by scanning its code and identifying links and names of input parameters. Their approach is context- and flow-insensitive, so it cannot capture distinct invocations along different paths of execution and may result in imprecise models. Licata and Krishnamurthi [13] use static analysis to accurately account for user operations in web applications, such as the use of the back button. Our two

techniques target different problems—their goal is to model and verify the control flow of a web application, whereas we are mainly focused on identifying parameter mismatches.

Several related approaches use program analysis to verify the HTML output of a web application [1, 3, 14]. They use program analysis to approximate the set of HTML pages that could be generated by an application, similar to what Step 2 of our approach does, and then check the pages using an HTML validator. These approaches are focused exclusively on validating the correct syntactical structure of the generated HTML pages and could not be used to identify parameter mismatches in web applications.

Closely related to these approaches is a technique by Gould, Su, and Devanbu [10], which performs a static verification of dynamically generated SQL queries. Their technique performs string analysis on a variable that contains an SQL database query, and then examines the possible queries to check that they are valid. At a high-level, this approach is somewhat similar to ours, but applied to SQL queries instead of interface invocations. Our analysis, however, is complicated by the fact that 1) the identification of interface invocations involves path-dependent properties and 2) the structure of valid accepted interfaces is not known a priori, as is the case for valid SQL queries.

Although web services are closely related to web applications, the work in the two areas has important differences. Web services have interfaces and interactions that are specified and described by languages such as BPEL and WSDL. Work on web services has focused on using these specifications to perform analysis and verification [8, 9, 15]. These types of specifications do not exist for web applications, and so techniques defined for web services are not directly applicable in our context. Instead, the interfaces and interactions must be derived from an analysis of the code.

6. CONCLUSION AND FUTURE WORK

We presented an approach for automatically identifying parameter mismatches in web applications. Preventing parameter mismatches is important because modern web applications use communication between their components to build pages that integrate content and data from multiple sources. Our technique works by (1) identifying the set of interfaces accepted by each web component, (2) determining the set of interface invocations performed by each web component, and (3) checking whether each interface invocation matches an accepted interface of the invocation target.

To evaluate our approach, we built a prototype implementation, , and ran it against four real web applications. was able to identify actual errors in all of the subjects. Although its efficiency could be improved, is efficient enough to be incorporated into existing quality assurance processes. Overall, we believe that the results of the evaluation are promising, indicate that our approach is effective and practical, and motivate further research.

Our immediate plans for future work involve addressing some of the limitations of our analysis. Particularly, we will implement and evaluate some of the optimizations discussed in Section 4.3 to improve 's efficiency. We will also extend our approach so that it accounts for JavaScript actions that can affect interface invocations and for servlet API functions that can redirect interface invocations or include imported code.

Acknowledgments

This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

7. REFERENCES

[1] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M. Ernst. Finding Bugs in Dynamic Web Applications. In , Jul. 2008.

[2] A. Betin-Can and T. Bultan. Verifiable Web Services with Hierarchical Interfaces. In , Jul. 2005.

[3] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static Validation of Dynamically Generated HTML. In , Jun. 2001.

[4] T. Bultan. Modeling Interactions of Web Software. In , Nov. 2006.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In , Jun. 2003.

[6] Y. Deng, P. Frankl, and J. Wang. Testing web database applications. , 29(5):1–10, 2004.

[7] S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel. Helping End-Users "Engineer" Dependable Web Applications. In , Nov. 2005.

[8] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions In , 2003.

[9] X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Services In , 2004.

[10] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In , May 2004.

[11] W. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In , Nov. 2006.

[12] W. G. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In , Sep. 2007.

[13] D. Licata and S. Krishnamurthi. Verifying Interactive Web Programs. In , Sep. 2004.

[14] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In , May 2005.

[15] S.K. Rajamani and J. Rehof. Models for Contract Conformance In , 2004

[16] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In , May 2001.