Domain and Value Checking of Web Application Invocation Arguments

William G.J. Halfond
University of Southern California
Los Angeles, California, USA
Email: halfond@usc.edu

***Abstract*—Invocations are widely used by many web applications, but have been found to be a common source of errors. This paper presents a new technique that can statically verify that an invocation's set of argument names, types, and request method match the constraints of a target interface. An empirical evaluation of the technique shows that it is successful at identifying previously unknown errors in web applications.**

I. Introduction

Components of modern web applications communicate among themselves extensively to generate customized and integrated content. This communication is done by sending Hyper-Text Transfer Protocol (HTTP) based invocations that contain arguments and data. Invocations are widely used to implement web forms, hyperlinks, and to send AJAX-based data to servers. Errors in invocations are one of the most common errors made by web application developers [1]. In early simple web applications, web crawlers could easily identify these types of errors. However, modern web applications dynamically generate HTML pages. Web crawlers cannot provide any guarantees that they will visit all of the pages generated by an application and could therefore miss errors. More recent techniques propose analyses for estimating the HTML pages generated by an application and then check these using an HTML validator [2], [3]. However, these approaches can only check for syntactic correctness of an invocation and do not check that the invocation matches the target's interface. Previous work by the author [4] uses static analysis based techniques to verify the correctness of invocations, but can only catch errors in which the set of names of an interface and invocation do not match. More subtle, but equally impactful, errors related to the value of the arguments in an invocation are undetectable by prior work.

In this paper, the PI presents a new approach for checking web application invocations that addresses the limitations of prior work. The new approach introduces novel analyses that allow it to check for additional correctness properties related to domain constraints on argument values and request methods. To do this, the approach analyzes the code of the web application to identify substrings that will be used to form parts of the application's HTML output that define invocations. Based on the source of the substring, the approach is able to infer additional useful information about the invocation. This paper also presents the results of an empirical evaluation that measures the proposed approach in terms of its effectiveness at identifying invocation errors. The results of the evaluation are positive and indicate that the proposed approach is able to discover new errors in the subject applications with a low false positive rate.

II. Background Information and Example

When an HTTP message contains arguments to be consumed by the target component, it is referred to as an *invocation*. Examples include web forms, hyperlinks with parameters, and direct API calls. Arguments in an invocation are represented as name-value pairs. HTTP supports several *request methods* for encoding and transferring arguments to the target component, but the two most widely used are GET and POST.

Components receive an invocations via an *interface*, which is defined as the set of arguments accessed along a path of execution in a web application. Although all argument values are returned as strings, the use of certain types of operations on the returned value can be used to infer domain constraints. For example, calling `Integer.parse()` on the value of an argument implies that the argument is expected to be of type `Integer`. The set of such domain constraints placed on an interface along a specific path of execution is referred to as an *interface domain constraint* (IDC). An interface can have more than one IDC if different operations are performed on its arguments along different paths.

Figures 1 and 2 show the partial implementation of a web application. The servlet `OrderStatus` generates two different invocations that are encoded as web forms. The target of these invocations is servlet `ProcessOrder`, which is shown in Figure 2.

In the example web application there are three invocation related errors. The first error is that one of the values specified for the "task" hidden field at line 19 of `OrderStatus` does not match the value checked for at lines 14 or 17 of `ProcessOrder`. This error causes a silent failure and the order is not submitted. The second error occurs if the user specifies an alternate shipping address. This causes a number format exception at line 4 of `ProcessOrder` since it is assumed that all the options are represented by numbers even though line 10 of `OrderStatus` provides an alphanumeric value for the argument. The third error is that there is no case

978-1-4577-1639-3/11/$26.00

```
void service(Request req) {
 2. print("<html><body><h1>Confirm Order</h1>");
 3. String oid = req.getParam("oid");
 4. int quant = getQuantity(oid);
 5. print("<form method=POST action='ProcessOrder'>");
 6. print("<input type=hidden value="
          + oid + " name=oid>");
 7. print("<select name=shipto>");
 8. print("<option value=0>Billing Addr.</option>");
 9. print("<option value=1>Home Address</option>");
10. print("<option value=other>Alt.</option>");
11. print("</select>");
12. print("If other: <input type=text name=other>");
13. if(canModify(oid)) {
14.   print("<p>Enter new quantity: </p>");
15.   print("<input type=text name=quant value="
            + quant + ">");
16.   print("<input type=hidden value=modify "
            + "name=task>");
17.   print("<input type=submit value='Change"
            + " Quantity'>");
18. } else {
19.   print("<input type=hidden value=confirm "
            + "name=task>");
20.   print("<input type=submit value='Purchase'>");
21. }
22. print("</form></body></html>");
}
```

Fig. 1. Invoking component, `OrderStatus`.

```
void doPost(Request req) {
 2. String oid = req.getParam("oid");
 3. String task = req.getParam("task");
 4. int shipOption =
      Integer.parse(req.getParam("shipto"));
 5. String address=req.getParam("other");
 6. switch (shipOption) {
 7.   case 1:
 8.     address = getHomeAddress(oid);
 9.     break;
10.   case 2:
11.     saveOtherAddress(oid, address);
12.     break;
13. }
14. if(task.equals("purchase")) {
15.   submitOrder(oid, address);
16. }
17. if(task.equals("modify")) {
18.   int quant = Integer.parse(req.getParam("quant"));
19.   modifyOrder(oid, quant);
20.   submitOrder(oid, address);
21. }
}
```

Fig. 2. Target component, `ProcessOrder`.

that can handle when the billing address option is chosen at line 8 of `OrderStatus`.

III. Proposed Approach

The goal of the proposed approach is to statically check web application invocations for correctness and detect errors such as those illustrated in Section II. There are three basic steps to the approach (A) identify generated invocations, (B) compute interfaces and domain constraints, and (C) check that each invocation matches an interface.

A. Identify Invocation Related Information

The goal of this step is to identify invocation related information in each component of the web application. The information to be identified is: (a) the set of argument names that will be included in the invocation, (b) potential values for each argument, (c) domain information for each argument, and (d) the request method of the invocation. The general process of this step is that the approach computes the possible HTML pages that each component can generate. During this process, domain and value information is identified by tracking the source of each substring in the computed set of pages. Finally, the computed pages and substring source information are combined to identify the invocation information.

$$\texttt{Gen}[n] = \begin{cases} \{\{\}\} & \text{if n is method entry} \\ \{n\} & \text{if n generates output} \\ \{n\} & \text{if n is a callsite} \\ & \text{and target(n) has a summary} \\ \{\} & \text{otherwise} \end{cases}$$

$$\texttt{In}[n] = \bigcup_{p \in pred(n)} \texttt{Out}[p]$$

$$\texttt{Out}[n] = \{p | \forall i \in \texttt{In}[n], p \leftarrow \texttt{append}(i, \texttt{Gen}[n])\}$$

$$\texttt{fragments}(m) = \left\{ p | \forall s \in \texttt{Out}[exitNode(m)] \prod_{n \in s} \texttt{resolve}(n) \right\}$$

Fig. 3. Data-flow equations for HTML page extraction.

1) Compute Possible HTML Pages: The approach analyzes a web application to compute the HTML pages each component can generate. Prior work by the author [4] is extended, as described in Sections III-A2 and III-A3, to compute these pages in such a way as to preserve domain information about each invocation. For space considerations, the algorithms that serve as the starting point for the approach are summarized in Figure 3. The approach computes the fixed point solution to the data-flow equations and at the end of the computation, the fragment associated with the root method of each component contains the set of possible HTML pages that could be generated by executing the component.

2) Identify Domain and Value Information: The approach identifies domain and value information for each argument in an invocation. The key insight for this part of the approach is that the source of the substrings used to define invocations in an HTML page can provide useful information about the domain and possible values of each argument. For example, if a substring used to define the value of an invocation originates from a call to `StringBuilder.append(int)`, this indicates that the argument's domain is of type integer. To identify this type of information, strings from certain types of sources are identified and annotated using a process similar to static tainting. Then the strings and their corresponding annotations are tracked as the approach computes the fixed point solution to the equations in Figure 3.

The mechanism for identifying and tracking string sources starts with the `resolve` function, which analyzes a node n in an application and computes a conservative approximation of the string values that could be generated at that node. The general intuition is that when the `resolve` function analyzes a string source that can indicate domain or value information, a special *domain and value* (DV) function is used to complete the analysis. The DV function returns a finite state automaton (FSA) defined as the quintuple $(\Sigma, S, s_0, \delta, F)$ whose accepted language is the possible values that could be generated by the expression. In addition, the DV function also defines two

functions: $D : S \times \Sigma \rightarrow T$ that maps each transition to a domain type, where T is a *basic type* of character, integer, float, long, double, or string; and $V : S \times \Sigma \rightarrow \Sigma \cup *$ that maps each transition to a symbol in Σ or a special symbol $*$ that denotes any value. D is used to track the inferred domain of a substring and V is used to track possible values. A DV function is defined for each general type of string source. For the purpose of the description of the DV functions below, e refers to any transition $(S \times \Sigma)$ defined by δ and the function $L(e)$ returns the symbol associated with the transition e.

Functions that return a string variable: Substrings originating from these types of functions can have any value and a domain of string. This is represented as $V(e) = *$ and $D(e) = string$.

String constants: The string constant provides a value for the argument and a domain of string. This is represented as $V(e) = L(e)$ and $D(e) = string$.

Member of a collection: For example, a string variable defined by a specific member of a list of strings. More broadly, of the form $v = collection\langle T \rangle[x]$ where v is the string variable, $collection$ contains objects of type T, and x denotes the index of the collection that defines v. In this case, a domain can be provided based on the type of object contained in the collection. This is represented as $D(e) = T$, and $V(e) = collection[x]$ if the value is resolvable or $V(e) = *$ otherwise.

Conversion of a basic type to a string: For example, `Integer.toString()`. More broadly any function $convert(X) \rightarrow S$ where X is a basic type and S is a string type. This operation implies that the string should be a string representation of type X. This is represented as $D(e) = X$, and $V(e) = *$ if X is defined by a variable or $V(e) = L(e)$ otherwise.

Append a basic type to a string: For example, a call to `StringBuilder.append(int)`. More broadly, $append(S, X) \rightarrow S'$ where S is a string type, X is a basic type, and S' is the string representation of the concatenation of the two arguments. In this case, the domain of the substring that was appended to S should be X. This is represented as $D(e_X) = X$. $V(e_X) = *$ if X is defined by a variable or $V(e_X) = L(e_X)$ otherwise. The subscripts denote the subset of transitions defined by the FSA of the string representation of X.

3) Combining Information: The final part of identifying invocation related information is to combine the information identified by computing the HTML pages and the domain and value tracking. The key insight for this step is that substrings of the HTML pages that syntactically define an invocation's value will also have annotations from the DV functions. To identify this information, a custom parser is used to parse each of the computed HTML pages and recognize HTML tags while maintaining and recording any annotations.

Example: Using the equations listed in Figure 3, the `Out[exitNode]` of servlet `OrderStatus` is equal to {{2, 5–12, 14–17, 22}, {2, 5–12, 19–22}. The analysis performs `resolve` on each of the nodes in each of the sets that comprise `Out[exitNode]`. Nodes 2, 5, 7–12, 14, 16, 17, 19, 20, and 22 involve constants, so `resolve` returns the values of the constants and the domain information is any string (*). Nodes 6 and 15 originate from special string sources. The variable `oid` is defined by a function that returns strings and can be of any value (*), and the variable `quant` is an append of a basic type, so it is marked as type `int`. After computing the `resolve` function for each of the nodes, the final value of `fragments[service]` is comprised of two web pages, which differ only in that one traverses the true branch at line 13 and therefore includes an argument for `quant` and a different value for `task`

TABLE I
INVOCATIONS GENERATED BY SERVLET ORDERSTATUS.

#	Arguments
1	<oid, *, “”> <task, *, “modify”> <shipto, *, 0> <other, *, “”> <quant, INT, “”>
2	<oid, *, “”> <task, *, “modify”> <shipto, *, 1> <other, *, “”> <quant, INT, “”>
3	<oid, *, “”> <task, *, “modify”> <shipto, *, “other”> <other, *, “”> <quant, INT, “”>
4	<oid, *, “”> <task, *, “confirm”> <shipto, *, 0> <other, *, “”>
5	<oid, *, “”> <task, *, “confirm”> <shipto, *, 1> <other, *, “”>
6	<oid, *, “”> <task, *, “confirm”> <shipto, *,“other”> <other, *, “”>

The approach then parses the HTML to identify invocations. By examining the annotations associated with the substring that defines each argument's value, the value for arguments `oid` and `quant` are identified. The `<select>` tag has three different options that can each supply a different value. So three copies are made of each of the two web form based invocations. Each copy is assigned one of the three possible values for the `shipto` argument. The final result is the identification of six invocations originating from `OrderStatus`. These are shown in Table I. Each tuple in the table lists the name, domain type, and values of the identified argument.

B. Identify Interfaces

This step of the proposed approach identifies interface information for each component of a web application. The proposed approach extends prior work in interface analysis [5] to also identify the HTTP request method for each interface. The specific mechanism for specifying HTTP request methods depends on the framework. In the Java Enterprise Edition (JEE) framework, the name of the entry method first accessed specifies its expected request method. For example, the `doPost` or `doGet` method indicates that the POST or GET request methods, respectively, will be used to decode arguments. The proposed approach builds a call graph of the component and marks all methods that are reachable from the specially named root methods as having the request method of the originating method.

Example: `ProcessOrder` can accept two interfaces due to the branch taken at line 17: (1) {oid, task, shipto, other} and (2) {oid, task, shipto, other, quant}. From the implementation of `ProcessOrder` it is possible to infer domain information for some of the parameters. From this information, the first interface is determined to have an

TABLE II
ERRORS AND FALSE POSITIVES REPORTED BY WAIVE+ .

Subject	# Invk.	Confirmed Errors			False Positives		
		R.M.	N.	IDC	R.M.	N.	IDC
Bookstore	92	0	12	0	0	0	1
Daffodil	103	0	1	28	0	12	0
Filelister	14	0	3	4	0	0	0
JWMA	27	0	17	4	0	7	0
Total	236	0	33	36	0	19	1

IDC of int(shipto)∧(shipto=1∨shipto=2)∧task="purchase"; and the second interface has an IDC of int(shipto)∧(shipto=1∨shipto=2)∧task="modify"∧int(quant). Unless otherwise specified, the domain of a parameter is a string. Lastly, by traversing the call graph of `ProcessOrder` all parameters (and therefore, all interfaces) are identified as having originated from a method that expects a POST request.

C. Verify Invocations

The third step of the approach checks each invocation to ensure that it matches an interface of the invocation's target. An invocation matches an interface if the following three conditions hold: (1) the request method of the invocation is equal to the request method of the interface; (2) the set of the interface's parameter names and the invocation's argument names are equal; and (3) the domains and values of the invocation satisfy an IDC of the interface. For the third condition, domain and value constraints are checked. The domain of an argument is considered to match the domain of a parameter if both are of the same type or if the value of the argument can be successfully converted to the corresponding parameter's domain type. For example, if the parameter domain constraint is Integer and the argument value is "5," then the constraint would be satisfied.

Example: Consider the interfaces identified in Section III-B and the invocations shown in Table I. Each of the six invocations is checked to see if it matches either of the two interfaces. Only invocation 2 represents a correct invocation and the rest will be identified as errors.

IV. EVALUATION

The evaluation measures the precision of the reported results. The proposed approach was implemented as a prototype tool, WAIVE+. The subjects used in the evaluation are four Java Enterprise Edition (JEE) based web applications: Bookstore, Daffodil, Filelister, and JWMA. These applications range in size from 8,600 to 29,000 lines of code. All of the applications are available as open source and are implemented using a mix of static HTML, JavaScript, Java servlets, and regular Java code.

To address the research questions, WAIVE+ was run on the four applications. For each application the reported invocation errors were inspected. Table II shows the results of inspecting the reported invocations. Each invocation error was classified as either a confirmed error or a false positive. Invocations in both classifications were also further classified based on whether the error reported was due to a violation of one of the correctness properties explained in Section III-C: the invocation did not match an interface because of an incorrectly specified request method (R.M.), the argument names did not match the parameter names of any interface of the target (N.), and the value and domain information of an invocation did not match the interface domain constraint (IDC). The table also reports the total number of invocations identified for each application (# Invk.).

As the results in Table II show, WAIVE+ identified 69 erroneous invocations and had 20 false positives. Prior approaches can only detect errors related to names, so the comparable total of errors for WAIVE was 33 erroneous invocations and 19 false positives. These results indicate that the new domain information checks resulted in the discovery of 36 additional errors and 1 false positive. Overall, the results are very encouraging. The approach identified 36 new errors that had been previously undetectable while only producing one additional false positive.

V. CONCLUSIONS

This paper presented a technique for statically checking the correctness of web application invocations. The technique has three main steps: (1) identify the interfaces of a web application, (2) compute the invocations that can be generated by each component of the web application, and (3) check to ensure that each invocation matches an interface of the target component. An invocation matches an interface when (1) its HTTP request method matches that of the interface, (2) both reference the same set of named arguments, and (3) the value of each argument matches the domain constraints of the corresponding parameter. The approach was implemented and evaluated against several subject web applications. The results indicate that the approach is able to find additional errors related to the domain constraints of the interfaces while having a low false positive rate.

REFERENCES

[1] S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel, "Helping End-Users "Engineer" Dependable Web Applications," in *Proceedings of the International Symposium of Software Reliability Engineering*, November 2005, pp. 22–31.

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding Bugs in Dynamic Web Applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, July 2008.

[3] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," in *Proceedings of the International World Wide Web Conference*, May 2005, pp. 432–441.

[4] W. G. Halfond and A. Orso, "Automated Identification of Parameter Mismatches in Web Applications," in *Proceedings of the Symposium on the Foundations of Software Engineering*, November 2008.

[5] W. G. Halfond and A. Orso, "Improving Test Case Generation for Web Applications Using Automated Interface Discovery," in *Proceedings of the Symposium on the Foundations of Software Engineering*, September 2007.