2012 IEEE 23rd International Symposium on Software Reliability Engineering

Automated Checking of Web Application Invocations

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

***Abstract*—HTTP based invocations allow web application components to communicate among themselves and build dynamic customized web pages. Invocations are widely used by web applications, but are a common source of errors. Existing techniques are only able to verify limited correctness properties of web application invocations and omit key properties, such as an argument's type and value must match its target parameter's domain. This paper presents the first approach for verifying these correctness properties of web application invocations. An empirical evaluation of the technique shows that it is able to identify, with high precision, over 30% more invocation errors than were previously identified and that the approach has a low analysis runtime cost.**

***Keywords*-program analysis; web applications; invocations; verification;**

I. Introduction

Modern web applications integrate information from multiple sources in order to provide users with rich and customized content. To do this, the components of a web application *invoke* each other by sending arguments and requests for data via the Hyper-Text Transfer Protocol (HTTP). This style of component interaction is widely used to submit data via web forms, encode arguments in hyperlinks, transmit XML data to the server in AJAX-based web applications, and facilitate direct component to component communication. Unfortunately, errors in invocations are one of the most common types of errors [1] — developers omit required arguments, provide illegal values for an argument, or use the wrong HTTP method. These types of errors can cause a range of runtime failures, including null pointer exceptions and erroneous output.

Conceptually, checking a web application invocation is similar to how a compiler verifies an invocation of a method or procedure; the names and domains of arguments must match the invocation's target interfaces. However, compilers are unable to perform this check for web application invocations because the invocations are defined in dynamically generated object programs (i.e., HTML and HTTP) as opposed to the general purpose language of the application (e.g., Java). Even for techniques oriented towards web applications, characteristics of modern web applications make this problem challenging. Modern web applications can dynamically generate HTML pages and HTTP messages, which may both contain invocations, by concatenating hard-coded strings along different paths. Not only must a checking technique determine the set of dynamically generated invocations, but it must also infer the arguments' domain information, such as types and legal values, since the HTML pages and HTTP messages do not explicitly encode this information.

These complications make it difficult to detect invocation errors. Traditional web application verification techniques, such as web crawlers [2], [3], [4], are incomplete because they cannot provide guarantees that their crawling will identify all invocations generated by the application. Even if they could identify all invocations, they can only check the HTML string-based representation, which lacks explicit domain information. Recent techniques can statically estimate and then validate an application's HTML output (e.g., [5], [6]), but these only check the syntactic form of an invocation. The use of frameworks, such as JSON in AJAX-based web applications, can prevent errors at runtime, but does not provide a way to statically identify errors. Previous work by the author checks one aspect of invocation correctness — that the names specified by an invocation match one of its target interfaces [7]. However, it is not able to verify more advanced correctness properties related to the domain and encoding of the argument values.

This paper presents the first approach for verifying the correctness of the types and values of arguments defined in dynamically generated web application invocations. The approach computes the names, types, and values of the arguments of each invocation generated by a web application and verifies that each invocation matches an interface of its target in terms of the arguments provided, the domain of each argument, and the method used to encode the set of arguments. The approach uses static analysis based techniques to estimate the set of invocations generated by a web application and infer arguments' domain information. The empirical evaluation shows that the approach discovers over 30% more invocation errors than a state of the art prior approach. Moreover, these results were achieved with a low false positive rate and a reasonable analysis runtime cost.

The paper is organized as follows: In Sections II and III, the paper introduces background information and a small example web application that is used to illustrate the approach. The approach is presented in Section IV. Section V reports on the results of the empirical evaluation. Lastly, Sections VI and VII discuss related work and summarize the findings.

1071-9458/12 $26.00 © 2012 IEEE
DOI 10.1109/ISSRE.2012.37

invocation	=	⟨target, method, argument$^+$⟩
method	=	GET | POST
argument	=	⟨name, type, value⟩
type	=	* | INT | FLOAT | DOUBLE |...

Figure 1. Abstract representation of an invocation.

II. Background Information

A *web application* is a software system that is available to end users via the Internet. The basic modules of a web application are referred to as its *components*. These can be implemented in many different programming languages, such as Java and PHP. Each component provides a *root method*, which is the first method called when the component executes.

Components receive invocations via their interfaces. In a web application, an *interface* is defined as the set of arguments accessed along a path of execution in the web application. Each component accesses its arguments by calling a special API function, which takes the name of the argument and then, based on the HTTP request method, searches for the corresponding value in either the message header or body. 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 parameters along different paths.

When an HTTP message contains arguments to be consumed by the target component, it is referred to as an *invocation*. Depending on the implementation mechanism of an invocation, the invocation is referred to as direct or indirect. To perform a *direct invocation*, a component passes arguments to an API method that invokes the target component and returns the response. A component performs an *indirect invocation* when it creates an HTML tag, such as a web form or hyperlink, that allows the browser to send the invocation when the user performs a certain action (e.g., clicks on a submit button). HTTP supports several *request methods* for encoding and transferring arguments to the target component, but the two most widely used are GET and POST. For both, arguments are represented as name-value pairs. The former specifies that the pairs must be in the message headers as part of the URL string, while the latter specifies that the pairs will be in the HTTP message body. Figure 1 shows the definition of an invocation used in this paper. An invocation has a target component, request method, and set of arguments, each of which has a name, value, and type. In this paper, the value and type of an argument are referred to together as its *domain*. An *invocation error* occurs when the names and domains of the arguments provided in an invocation do not match an interface and IDC of the invocation's target component.

```
void _jspService(Request req)
 1. print("<html><body>");
 2. print("<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. print("</form></body></html>");
```

Figure 2. Invoking component, `OrderStatus`.

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

Figure 3. Target component, `ProcessOrder`.

III. Example Web Application

Figures 2 and 3 show excerpts from an online bookstore application that allow users to track and manage their orders. Both of the components shown in the figures are implemented as servlets, which is the Java Enterprise Edition's equivalent of a component. The servlet `OrderStatus` displays a page that allows users to update and confirm their order. When a user accesses the servlet, execution begins in the root method, `_jspService`. Line 3 retrieves the

value of the order ID argument, "oid," and line 4 accesses the quantity associated with the order item. Line 5 begins the display of the web form, which specifies the request method of POST and that the target of the form submission is `ProcessOrder`. Line 6 creates a hidden field, a type of field often used to maintain state information, that contains the order ID. Then lines 7–12 create a drop down box and text input field that prompts the user to select a shipping address. If the user is allowed to modify the order, then lines 14–17 display a text input field that allows the user to update the quantity and submit the order. Otherwise, lines 19–20 display only a submit button. Both paths also define a hidden field (lines 16 and 19) that specifies the action to be taken on the server side when the invocation is received.

When the form from `OrderStatus` is submitted, `ProcessOrder` (shown in Figure 3) is invoked. Lines 1 and 2 retrieve the hidden fields that specify the order ID and the action to be performed. Line 3 retrieves the value corresponding to the shipping address specified and casts it to an integer so it can be used to set the shipping address in the switch statement in lines 5–11. Depending on the action to be taken by `ProcessOrder`, the order is either submitted at line 13, or the quantity is updated before the order is submitted at lines 15–17.

In the example web application there are three invocation errors. The first error is that one of the values of the "task" hidden field, "confirm", generated by line 19 of `OrderStatus`, does not match the values checked for at lines 12 or 14 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 3 of `ProcessOrder` since it is assumed that all the options are represented by numbers even though line 10 of `OrderStatus` provides an alphanumeric value, "other", for the argument. The third error is that there is no case in the switch statement at lines 5–11 of `ProcessOrder` that can handle the "0" billing address option created by line 8 of `OrderStatus`.

Current techniques may be able to expose some of these errors, but have limitations which make it likely that many will be unidentified. Crawling techniques cannot identify the domain constraints on the parameters or argument types; HTML validators only report syntax errors; a mutation testing based approach [8] only targets a subset of the possible invocation errors; and prior work [7] only checks the names of the argument defined in the invocation.

IV. Checking Invocations

The goal of the approach is to statically verify the correctness of web application invocations by detecting errors such as those shown in Section III. At a high-level, the approach must (A) compute invocation information: targets, methods, and arguments, (B) identify interfaces and interface domain constraints, and then (C) verify that each invocation matches an interface of its target. Each of these steps is explained in more detail in the following sections.

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

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

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

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

Figure 4. Data-flow equations for HTML page extraction.

A. Compute Invocation Information

The approach analyzes each component to identify the invocation information shown in Figure 1; namely, (i) names of arguments in an invocation, (ii) type information for each argument, (iii) values supplied for an argument, (iv) the request method of the invocation, and (v) the target of the invocation.

The approach computes an approximation of the set of HTML pages that can be generated by the application's components and analyzes each page to extract the invocation information. The key insight is that the source of the substrings used to define invocations in an HTML page can provide domain information about the invocation beyond the syntax based analysis. For example, in `OrderStatus`, manual inspection reveals that the argument named "quant" is always assigned an integer value (lines 4 and 15). To take advantage of this insight, the approach annotates substrings that originate from certain sources. As the HTML output is computed, these annotations are maintained with the portion of the HTML page they define. Then the HTML pages are parsed and substrings that define invocation related tags are analyzed to identify the information defined by the HTML syntax and the domain information defined by annotations attached to the substrings.

To compute a component's HTML pages, the approach analyzes the component's methods and creates a summary for each one that describes the strings the method contributes to the HTML output. To create a summary, the approach first groups HTML generating nodes that are along the same path in the method's control-flow graph (Section IV-A1). Then the approach analyzes each grouping to determine its generated HTML fragments and domain information (Section IV-A2). The HTML fragments identified are assigned as the method summary. The methods are processed in reverse topological order with respect to the component's call graph so that a method's summary is available before a method that calls it is analyzed. Methods involved in recursive calls

are grouped together and treated as one method.

1) Grouping HTML-Generating Nodes: The approach casts the correct groupings of a method m's HTML generating nodes as the solution to the iterative data-flow equations shown in Figure 4. First, the approach initializes `Gen[n]` for each node n in m's control-flow graph (CFG) based on whether it directly generates HTML data or calls a function that contains nodes that generate HTML data. Identifying whether a node generates HTML data is done based on the signature of each node's target invocation method and is known on a per web application framework basis. The `In` and `Out` sets, each of which is a set of ordered sets of CFG nodes, are updated until a fixed point is reached. Once this point is reached, each ordered set in the `Out` set of m's exit node represents a sequence of HTML generating nodes that can be traversed along some path through m.

The formulation shown in Figure 4 follows the standard data-flow framework formulation [9]. The iterations of computing the `In` and `Out` sets will converge on a fixed point because (1) the upper bound of each ordered set is the number of nodes in the CFG; (2) the size of the `In` and `Out` sets is upper bounded by the number of acyclic paths though m's CFG; and (3) each iterative computation of `In` and `Out` monotonically increases the size of the set or terminates if there are no further changes to any `Out` set. Note that the `In` set of the join node after a loop has at least one ordered set for the path traversing the branch that entered a loop and the branch that did not enter the loop. Important additional details on how the string computation handles loop unraveling are discussed below in Section IV-A2.

Example: Consider servlet `OrderStatus` in Figure 2. When the analysis terminates, the `Out` set of the exit node of `OrderStatus` is {{1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 21}, {1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 19, 20, 21}}. The first set represents the HTML-generating nodes along the path taking the true branch at node 13 and the second set is the corresponding false branch.

2) Identify Generated HTML Strings: To finish the computation of a method m's summary, the approach identifies the set of HTML pages produced by each ordered set in the `Out` set of m's exit node. This is done by iterating over each node n in each ordered set, calling `resolve(n)` to identify the output produced by n, and then appending this output together. The complexity in this step is in the `resolve` function, which computes the string values that a node can output and tracks the domain-indicating source information associated with those strings.

The input to the `resolve` function is a CFG node n that represents an HTML generating statement and the output is a finite state automaton (FSA) that represents the generated HTML strings and domain information about those strings. The FSA is defined as the tuple $\langle \Sigma, S, s_0, \delta, F, T, V \rangle$ where Σ is the alphabet consisting of all Unicode characters, special placeholder symbols P_i, and $*$ for any character; S is the set of states; s_0 is the initial state; δ is the transition function; and F is the set of final states. There are an additional two functions: $T : S \times \Sigma \rightarrow t$ that maps each transition to a type, where t is a *basic type* of character, integer, float, long, double, string, or variation thereof; and $V : S \times \Sigma \rightarrow \{\Sigma^*\}$ that maps each transition to a set of strings. T is used to track the inferred type of a substring and V is used to track possible values.

The `resolve` function performs an intra-procedural analysis of a node n and leaves placeholders in the summary for content that cannot be resolved by definitions within the method. For example, if the i^{th} parameter to the enclosing method of n is used to define a string value, the placeholder P_i is inserted and later, arguments at a callsite to the method are used to define P_i. The use of placeholders improves the context-sensitivity of the approach. There are three general cases handled by the `resolve` function based on whether the string value at n: (1) references a constant string; (2) defines a string expression (e.g., string value defined by a concatenation); or (3) invokes a method with a summary. In the first case, the possible constant strings are identified via constant propagation and the FSA representing those values is returned. In the second case, the `resolve` function leverages the Java String Analysis (JSA) [10] to compute a conservative estimate of the result of the string expression defining the string output of a node. For the third case, the method summary is used and the arguments at n are substituted for any placeholders in the summary.

HTML content generated by a node n within a loop is approximated by the approach. If the loop is over a possibly infinite set or the content is not dependent on the loop iterator, then the output is estimated by assuming the loop is executed once. However, if the HTML generated by n is dependent on the loop iterator and the loop is iterating over a statically defined set of size k (e.g., an array of string constants), then the FSA generates one string containing the k variations that could be generated at n. For example, the statement `print(''<option value=$A>'')` where `$A` is iterating over the set {"1", "2"}, generates the string `''<option value=1><option value=2>''`. Although this approximation loses sequencing information between nodes in a loop, it is effective for the approach because the exact ordering of arguments or invocations is not relevant for their correctness. This approximation only leads to errors if the generated HTML is dependent on the loop executing a specific number of times or the content generated by two or more nodes in the loop body must be concatenated together in order to be syntactically valid HTML. A path sensitive analysis (e.g., symbolic execution) could avoid this potential problem, but as the results in Section V show, there does not appear to be a need for this advanced technique.

The `resolve` function also tracks domain information for the identified string values. This information is maintained by the T and V functions defined in the output FSA.

The key intuition is that the source of a string analyzed by the `resolve` function, either when tracking the source of constant strings or approximating a string expression, can provide information about the type and/or values a substring can have. To define the T and V functions, a one-time effort must be performed for all methods of a language framework (e.g., Java Enterprise Edition) that fall into one of the categories discussed below. No application specific mappings are needed unless the application code bypasses the language framework to perform string operations. For the purpose of the description 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.

String constants: For example, a hard-coded string variable. The string constant provides a value for the argument and a domain of string. This is represented as $V(e) = L(e)$ and $T(e) = string$.

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 types of t, and x denotes the index of the collection that defines v. In this case, a domain can be provided based on the type (t) of object contained in the collection. If the analysis can resolve x and $collection$ is a statically defined set, then the identified member is used as the argument's value and $V(e) = collection[x]$. If x cannot be resolved, but $collection$ is a statically defined set, then the value is safely approximated as $V(e) = v \in collection$. Otherwise, $V(e) = *$. In all cases, $T(e) = t$.

Functions that return a string variable: For example, the `toString` method of Java objects. Substrings originating from these types of functions can have any value and a domain of string. This is represented as $V(e) = *$ and $T(e) = string$. Note that certain types of functions, explained below, are special-cased to give more precise domain information.

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 type of function specified that the string should be a string representation of type X. This is represented as $T(e) = X$, 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 approach can infer that the domain of the substring that was appended to S should be X. This is represented as $T(e_X) = X$, and $V(e_X) = *$ if X is defined by a variable or $V(e_X) = L(e_X)$ otherwise. Here the subscripts denote the subset of transitions in S' defined by the FSA of the string representation of X. The T and V values for the substring of S' defined by S remain the same.

Example: Consider again the `Out[exitNode]` set of servlet `OrderStatus` shown in Figure 2. To finish computing the summary for the method, the approach calls `resolve` on each of the nodes of the two ordered sets in `Out[exitNode]`. For nodes 1, 2, 5, 7–12, 14, 20, and 21, the `resolve` function identifies the string constants that define their generated HTML and would also determine that the domain information is any string (*). For nodes 16, 17, and 19, the `resolve` function computes the result of the string expression and returns the concatenation of the two constant strings. Here also the domain would be any string. Although nodes 6 and 15 are handled similarly, they also allow for the inference of type information. The string expressions at nodes 6 and 15 append the variables `oid` and `quant`, which originate from special string sources. The variable `oid` is defined by a function that returns strings, so the type is marked as string and the value is any string (*). The operation involving the variable `quant` is an append of a basic type (the `javac` compiler automatically converts part of line 15 of `OrderStatus` to `StringBuilder.append(int)`), so the substring represented by `quant` is marked as type `int`.

3) Invocation Extraction: Once all of the summaries have been computed, the approach analyzes the summary of the root method, which represents the HTML pages that could be generated by the component. The approach analyzes the summary to extract invocation information from the syntax of the invocation related HTML tags and also the annotations attached to the substrings that define these tags. The key insight is that if a particular substring is used to define an invocation related tag and it originated from a special string source, then the annotations can be used to augment the invocation information extracted from the HTML page. To identify and account for the additional information, the approach uses a customized HTML parser that can analyze the HTML representations in the summary and account for the annotations attached to invocation related HTML tags.

The input to the parser is the summary of the root method of the component. The output is a set of invocations, defined as shown in Figure 1. The parser analyzes the definition of each invocation related HTML tag to identify both the information derived from the syntax and from the annotations. The set of analyzed HTML tags includes all tags that contain a URL based attribute, since these could encode an invocation using the GET request method, and all form related tags, since these could encode either a GET or POST request method. Content within JavaScript tags is also analyzed for any URL based access, which could contain a GET based invocation. The content of URL attributes is analyzed with a customized URL parser to identify substrings that define the target, argument names, and the provided values. The corresponding substrings are also examined for domain information. For invocations defined by form related

Table I
INVOCATIONS GENERATED BY SERVLET ORDERSTATUS.

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

tags, each <form> tag is analyzed to identify the target and request method, then the associated input, select boxes, and text areas are analyzed to identify the names of the arguments defined, any default values provided, and the type and value annotations associated with each of the tags.

Lastly, the identification of information related to direct invocations is also performed at this time. The arguments of each call site that can be used to issue a direct invocation are analyzed using the resolve function. (The invocations are string based, so the arguments always map or convert to a string.) The strings computed by the resolve function are parsed to identify the invocation attributes. The method by which the strings are parsed depends on the request method employed by the specific API function.

Example: The approach parses the summary of _jspService in order to identify invocation information. As can be seen from the code listing, the two ordered sets will each generate one web form, which vary only in that one includes an argument for quant and a different value for task. The annotations associated with oid indicate that it is a string and can have any string value. The annotations for quant indicate that it is defined as an integer. Finally, the <select> tag has three different options that can each supply a different value, so the approach creates an invocation that represents each of the three possible values for the shipto argument (i.e., 0, 1, and “other”). The final result is the identification of six invocations originating from OrderStatus. These are shown in Table I.

B. Identify Interface Request Method

The second step of the approach identifies interface information for each component of a web application. To identify all of the information needed for the additional correctness properties, the approach extends existing work in interface and interface domain constraint identification, WAM [11], to also identify the expected request method. Although the interface identification is generally a framework-independent analysis, the specific mechanism for identifying the HTTP request methods does depend on the framework. Since the prototype implementation targets Java Enterprise Edition (JEE) based web applications, this section describes request method identification for that framework.

In the JEE framework, the name of the root method 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 complicating factor is that it is possible for servlets to define multiple such methods, each one implying a different request method. Control flow paths that originate in these methods can also merge. To address this, the proposed approach builds a call graph of the component and identifies all methods that are reachable from the specially named root methods of the component and then marks them as having the request method of the originating method. If a method is reachable from more than one such method, the method has multiple markings. During the interface analysis, all interface elements accessed within each method are marked as having the containing method’s request method. Later, the request method annotations of each interface’s parameters are analyzed to determine if an interface can be accessed via one or more different request methods.

Example: Consider again the servlet ProcessOrder in Figure 3. The WAM analysis determines that, depending on the branch taken at line 14, ProcessOrder can utilize one of two interfaces: (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. Lines 3 and 15 indicate that the two accessed parameters should both be of type Integer. The comparisons of “shipto” against “1” and “2” at lines 6 and 9, and the comparison of “task” against “purchase” and “modify” indicate that these are relevant values for the domain of each parameter. From this information, the first interface is determined to have an 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, the extension to WAM traverses the call graph of ProcessOrder and identifies that all parameters (and therefore, all interfaces) originated from a method that expects a POST request.

C. Verify Invocations

Once both the interfaces and invocation information has been identified, the approach checks that each invocation matches an interface of its target component. An invocation “matches” an interface when the following three conditions hold: (1) both contain equivalent sets of named parameters, (2) both have the same HTTP request method, and (3) the

inferred domain of an invocation's arguments satisfy an IDC of the interface.

To verify an invocation, the approach first retrieves all interfaces associated with the target of the invocation. For each interface, the approach checks the first two conditions; that the request method and set of arguments/parameters matches exactly. If these two conditions hold, then the approach checks the third condition. For each argument, its domain matches the domain of an interface's parameter if both are of the same type (e.g., String) 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 because `Integer.parseInt(``5'')` successfully converts the value to an Integer representation. The approach also checks linear arithmetic for numeric types and string equality.

Example: To illustrate, consider the interfaces identified in Section IV-B and the invocations shown in Table I. Invocation 1 matches the request method and names of interface 2. However, it fails because its value for "shipto" is "0" and that does not satisfy the constraint on the "shipto" parameter. Invocation 2 matches the request method and names of interface 2. The arguments also satisfy the domain constraints of the interface, so this invocation is a complete match. Invocation 3 matches the request method and names of interface 2. However, the domain for "shipto", which is any string, does not satisfy the parameter's integer domain constraint and the argument's value, "other", cannot be converted to an integer. Invocation 4 matches the request method and names of interface 1. However, the values of arguments "task" and "shipto" do not satisfy any of the constraints on the corresponding parameters. Invocation 5 matches the request method and names of interface 1, but the value of "task" does not satisfy the corresponding parameter's constraints. Invocation 6 matches the request method and names of interface 1. However, its value of "task" does not satisfy the constraint on the corresponding parameter and the domain of "shipto" does not match the corresponding parameter's constraint of integer and its value cannot be converted to an integer.

D. Additional Discussion

In general, the technique computes a safe (conservative) approximation of the invocations of a web application since the data-flow analysis and `resolve` function assumes all paths are feasible. This safety property holds as long as the application builds HTML pages and performs direct invocations with the APIs provided by the web application framework. Since performing these operation outside of the API is very complicated, this is a reasonable assumption. Nonetheless, there are three cases in which the safety of the invocation identification technique could be violated: (1) strings that define invocations are defined externally to the component's code, (2) JavaScript is used to dynamically modify portions of the HTML code that define invocations, and (3) JavaScript is used to directly generate an invocation.

For the first case, external strings that define invocations can come from a database or resource file. This method of generating HTML pages is used in several frameworks. The approach can handle this situation by special casing the `Gen` sets for call sites that access external strings and putting placeholders in for the content that could then be replaced later in the analysis process by the actual external content. Automating this process is one avenue of future work the author will explore. For the second and third cases, future work by the author will leverage new results in modeling the effect on invocations of dynamically generated JavaScript accessing the HTML DOM [12]. Note that for the subjects reported on in Section V, JavaScript did not alter the inferred argument domain information.

V. Evaluation

The evaluation measures the time necessary to run the invocation verification technique and the precision of its reported results. Since the approach is the first technique to verify invocation arguments' domain information, no direct comparison is possible. However, the evaluation does contrast the time and number of errors discovered against the most closely related approach, WAIVE [7], in order to provide a baseline for comparison. The following research questions are addressed in the evaluation:

RQ1: How much time is needed to run the technique?

RQ2: What is the approach's precision in identifying domain-related invocation errors?

RQ3: How many new errors are identified as compared to previously known errors?

A. Implementation

For the evaluation, the proposed approach was implemented as a prototype tool, ASCEND. The implementation of ASCEND is in Java and the tool analyzes the bytecode of web applications written using the Java Enterprise Edition (JEE) framework. To implement the interface analysis, ASCEND uses an extended version of the WAM interface analysis [11]. The computation of the HTML pages makes use of the Soot program analysis framework (http://www.sable.mcgill.ca/soot/) to generate call graphs and control-flow graphs, a modified version of JSA [10] to analyze string expressions, and a customized version of HTMLParser (http://htmlparser.sourceforge.net/) to parse HTML fragments. The checking algorithms in Section IV-C are written in Java.

B. Subjects

The evaluation is performed using seven Java based web applications. Three of these, Bookstore, Daffodil, and Filelister, were selected because they have been used in prior related work and the remainder were chosen to broaden the

Table II
SUBJECT APPLICATIONS FOR THE EVALUATION

Subject	*Description*	*LOC*	*Cls.*
Bookstore	Online bookstore.	19,218	29
Classifieds	Ad mgmt.	11,203	19
Daffodil	Customer mgmt.	19,236	121
Empldir	Employee DB front end	5,823	10
Events	Track online events	7,327	13
Filelister	Online file browser.	8,773	42
Portal	Club front end	16,849	28

subject pool and include more applications that were developed using web application frameworks. All of the subjects' implementations are a mix of static HTML, JavaScript, Java servlets, framework libraries, and regular Java code. The implementations include HTML generated within loops and inter-procedurally. The subjects' source code is available via the author's website.[1] Table II provides each subject's lines of code (*LOC*) and class (*Cls.*) count. The LOC includes only the Java code that defines servlets, JSP pages, and back end server code.

C. Experiment Execution

To obtain timing measurements, WAIVE and ASCEND were run on each of the subject applications. The machine used was an Intel Quad-Core E5606@2.13Ghz with 16GB DDR3 SDRAM running Ubuntu 12.04 with 8GB RAM dedicated to the JVM heap. For each application, Figure 5 shows the time, in seconds, each approach took. This measurement is further divided to show the time to run the interface analysis and the invocation analysis. Verification time is not shown because it was negligible (1-2 seconds) for all applications.

Table III shows the data for the second and third research questions. The column labeled "# Invk." shows the number of unique invocations for each application. This number is higher than previously reported numbers for the subjects [7], because an invocation is considered unique if it has different values for an argument, instead of different argument names. The column labeled WAIVE shows the number of errors identified by the approach. The columns under ASCEND show the number of confirmed domain-related errors (D_c), the number of false positive domain-related errors identified (D_{fp}), name related errors identified (N), and the total errors identified by ASCEND, excluding false positives (T). To classify the errors, each invocation was manually inspected and classified. No errors for request method types were found in the subject applications, so no count is reported for that category. All of the errors occurred naturally in the application and were not seeded.

D. Discussion of Results

For **RQ1**, Figure 5 shows that the total runtime of ASCEND was, on average, 50% higher than WAIVE for all

[1] http://www-bcf.usc.edu/~halfond/testbed.html

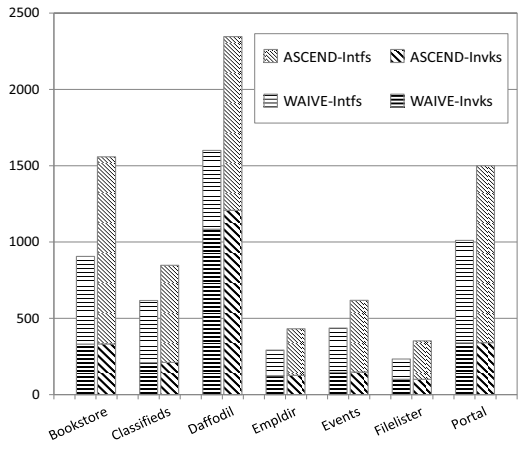

Figure 5. Timing results (s) of WAIVE and ASCEND

subject applications. Most of ASCEND's additional analysis time came from the average 88% increase in interface analysis time as opposed to an average 6% increase in invocation extraction time. The interface analysis time increased because ASCEND must also compute an application's IDCs to check argument domains. The IDC analysis is expensive because it computes the data-flow of every parameter in every interface. Invocation extraction time increased less because the overhead associated with identifying string sources and propagating that information is negligible compared to the computations that must already be done to identify argument names. Overall, the analysis time for ASCEND ranged from six to forty minutes, with an average of about eighteen minutes. This analysis time, while not trivial, is certainly reasonable and would not be a barrier to usage.

For **RQ2**, Table III shows the number of confirmed and false positive domain-related error reports generated by ASCEND. In total, ASCEND identified 106 domain-related errors, of which 8 were false positives, a roughly 7% false positive rate. The errors and false positives were investigated in order to determine their root cause. Two root causes dominated the true errors: (1) string arguments provided for parameters that were supposed to be an integer type, and (2) incorrect values provided for arguments where the domain constraint required one of a specific set of values. The second case occurred when an application provided a set of possible values for a drop down box and allowed the user to select among them, but did not handle one case, which led to an exception. Inspection of the code made it clear that this was an error and not the intended behavior. The false positives were almost all due to IDCs that were incorrectly identified. This occurred for two reasons. The first was the use of an input sanitization routine that would check input parameters differently based on their intended use. Since the data-flow analysis assumed all paths were feasible, the extra paths that performed other checks were also analyzed and these added spurious constraints that caused an otherwise

Table III
ERRORS IDENTIFIED BY WAIVE AND ASCEND.

Subject	# Invk.	WAIVE	ASCEND			
			D_c	D_{fp}	N	T
Bookstore	140	82	24	1	82	106
Classifieds	99	78	9	2	78	87
Daffodil	105	13	28	0	13	41
Empldir	36	19	5	2	19	24
Events	55	30	9	2	30	39
Filelister	16	3	4	0	3	7
Portal	151	110	19	1	110	129
Total	602	335	98	8	335	433

legal invocation to fail the check. The second reason was that, to check certain inputs, developers would convert the input to another type (e.g., `long` or `double`) even when that was not the intended type of the parameter. Nonetheless, this would add that constraint to the IDC and cause the check to fail. It is worth noting that more advanced techniques for interface identification, such as symbolic execution, could be used to reduce the first reason for false positives, which was also the dominant cause.

The experiments did not calculate false negatives due to the difficulty in accurately and manually calculating the string output along each path of the web application. False negatives could occur in the following scenarios: (1) the interface analysis generates a spurious interface (i.e., one along an infeasible path) and an erroneous invocation matches that interface; (2) JavaScript modifies an invocation at runtime in such a way that it leads to an error; or (3) the HTML surrounding an erroneous invocation contains syntax errors from which the parser cannot recover and therefore cannot extract the invocation. The first two are addressable by future work and the third is handled by flagging unparseable HTML, which is then manually inspected.

For the subjects in the evaluation, no errors were found in the HTTP request method encoding. Since these errors always lead to an observable and complete runtime failure of an invocation, it is unlikely that they would still be present in web applications that have been, even lightly, tested. The author hypothesizes that these types of errors are more likely to occur in the initial development of a web application and, therefore, this type of check would be more useful at that time.

For **RQ3**, the evaluation compares the number of errors found by ASCEND against the number found by WAIVE. The set of errors discovered by ASCEND was always a superset of those found by WAIVE. ASCEND found 433 errors compared to 335 found by WAIVE, an almost 30% increase in identified errors. Both approaches found the same set of name related errors, so this increase is completely due to the additional domain checks introduced by the ASCEND approach.

E. Threats to Validity

The design of the evaluation suggests two threats to validity, which the author has tried to address. First, to increase external validity, the choice of subjects contains problematic, but widely used, features, such as JavaScript, HTML generated inter-procedurally and within loops, use of web application frameworks, and input from external sources. Second, the approach is compared against prior work also by the author instead of independent tools. Although comparison against an independent tool is preferable, ASCEND represents the first technique to check for argument domain correctness and WAIVE is the most closely related prior approach.

VI. RELATED WORK

A preliminary version of the results and techniques presented in this paper appeared as a four page shortpaper [13]. This paper presents an extended empirical evaluation, more complete description of the technique, related work, and an illustrated example.

Prior work by the author also used static analysis to verify the correctness properties of web application invocations [7]. As compared to this approach, there are several conceptual differences. The primary conceptual difference is that the new approach is able to detect errors related to the domains, as opposed to only the names, of the arguments supplied in an invocation. Another is that the new approach uses a different formulation of the HTML page estimation algorithms and `resolve` function to enable domain information tracking. Additional differences are the extensions to the interface identification, which now also identifies the request method, and the verification process, which introduces new correctness semantics related to the arguments' types and values.

Recent approaches have used web crawlers to build models of web applications and then check these models for various correctness properties [14], [15], [4]. Two advantages for web crawling are that they can accurately handle JavaScript encountered during the crawl and will work if the implementation is not available. However, the disadvantages are that, given the source code, static analysis techniques will be more complete [11], interface specifications must still be provided, and the string based representation seen by the crawlers lacks the additional type and value information identified by ASCEND. These disadvantages limit the correctness properties that can be checked and the completeness of any checks. Capture replay techniques [16] have similar advantages and disadvantages as web crawling based techniques.

Several approaches also leverage static analysis to verify certain correctness properties of web applications. One approach performs static verification of web applications that were written using the `<bigwig>` framework, but is not easily generalizable to other language frameworks [17]. An

approach by Minamide [6] uses a context-free grammar to approximate and then validate a component's output, and Apollo uses concolic execution to model a web component's output [5]. Both of these approaches only check for syntactic validity and cannot check other properties related to invocation correctness. Lastly, static verification techniques for web services rely on WSDL and BPEL specifications, which are not provided nor applicable for the general type of web applications addressed in this approach.

Testing based approaches can also detect invocation errors. Several techniques generate test cases based on a web application model [18], [19], [20]. Their drawback is that these models must be manually specified, which is both time consuming and error-prone. Recent work in mutation testing [8] defines several new mutation operators that could target some, but not all, of the types of errors identified by ASCEND. More general test-input generation approaches [11], [21] can trigger invocation related errors, but lack a mechanism for identifying when an observed invocation is incorrect.

VII. Summary

This paper presented the first technique, ASCEND, for statically checking the correctness of the web application invocation arguments' domains. ASCEND uses static analysis to compute the interfaces and invocations generated by a web application and verifies that each invocation matches an interface of its target in terms of the arguments supplied, the domain of each argument, and the HTTP request method used for the parameter. The approach was compared against a prior invocation verification technique and the results were very encouraging. ASCEND was able to find 30% more errors in the subject web applications with a low false positive rate of 7%. Overall, the results indicate that ASCEND is an effective technique for identifying errors in web application invocations.

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] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically Testing Dynamic Web Sites," in *Proceedings the International World Wide Web Conference*, Honolulu, Hawaii, USA, May 2002.

[3] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," in *Proceedings of the International World Wide Web Conference*, Budapest, Hungary, May 2003, pp. 148–159.

[4] A. Mesbah and A. van Deursen, "Invariant-Based Automatic Testing of Ajax User Interfaces," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers*. IEEE Computer Society, May 2009, pp. 210–220.

[5] 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.

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

[7] 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.

[8] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *5th International Workshop on Mutation Analysis*. IEEE, April 2010, pp. 132–141.

[9] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise Analysis of String Expressions," in *Proceedings of the International Static Analysis Symposium*, San Diego, CA, USA, June 2003, pp. 1–18.

[11] 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*, Dubrovnik, Croatia, September 2007.

[12] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *SIGSOFT FSE*, 2011, pp. 59–69.

[13] W. G. J. Halfond, "Domain and value checking of web application invocation arguments," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2011, pp. 544–547.

[14] S. Elbaum, K.-R. Chilakamarri, M. F. II, and G. Rothermel, "Web Application Characterization Through Directed Requests," in *Proceedings of the International Workshop on Dynamic Analysis*, Shanghai, China, May 2006, pp. 49–56.

[15] M. Fisher, S. Elbaum, and G. Rothermel, "Dynamic characterization of web application interfaces," *Fundamental Approaches to Software Engineering*, pp. 260–275, 2007.

[16] S. Elbaum, G. Rothermel, S. Karre, and M. F. II, "Leveraging User-Session Data to Support Web Application Testing," *Transactions On Software Engineering*, vol. 31, no. 3, pp. 187–202, March 2005.

[17] C. Brabrand, A. Møller, and M. I. Schwartzbach, "The Bigwig Project," *Transactions on Internet Technology*, vol. 2, no. 2, pp. 79–114, 2002.

[18] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing Web Applications by Modeling with FSMs," *Software Systems and Modeling*, vol. 4, no. 3, pp. 326–345, July 2005.

[19] X. Jia and H. Liu, "Rigorous and Automatic Testing of Web Applications," in *Proceedings of the International Conference on Software Engineering and Applications*, Cambridge, Massachusetts, USA, November 2002, pp. 280–285.

[20] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," in *Proceedings of the International Conference on Software Engineering*, Toronto, Ontario, Canada, May 2001, pp. 25–34.

[21] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass Testing of Web Applications," *Proceedings of the International Symposium on Software Reliability Engineering*, vol. 0, pp. 187–197, 2004.