10-12-2006

Dynamic Characterization of Web Application Interfaces

Marc Randall Fisher II
University of Nebraska-Lincoln, fisherii@google.com

Sebastian Elbaum
University of Nebraska-Lincoln, selbaum@virginia.edu

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2006-0014
Issued Oct. 12, 2006
Published in Matthew B. Dwyer and Antónia Lopes, editors, Fundamental Approaches to Software Engineering, volume 4422/2007 of Lecture Notes in Computer Science, pages 260-275, Braga, Portugal, March 2007.

Dynamic Characterization of Web Application Interfaces

Marc Fisher II, Sebastian Elbaum, and Gregg Rothermel

University of Nebraska-Lincoln
{mfisher,elbaum,grother}@cse.unl.edu

Abstract. Web applications are increasingly prominent in society, serving a wide variety of user needs. Engineers seeking to enhance, test, and maintain these applications and third-party programmers wishing to utlize these applications need to understand their interfaces. In this paper, therefore, we present methodologies for characterizing the interfaces of web applications through a form of dynamic analysis, in which directed requests are sent to the application, and responses are analyzed to draw inferences about its interface. We also provide mechanisms to increase the scalability of the approach. Finally, we evaluate the approach's performance on six non-trivial web applications.

1 Introduction

Consider a flight reservation web application, such as Expedia. Such an application compiles data from multiple airlines, and provides a web site where customers can search for flights and purchase tickets. The site itself consists of HTML forms that are displayed to the customer in a web browser. Within these forms, the customer can enter information in fields (e.g. radio buttons, text fields) to specify the parameters for a flight (e.g. departure date, return date, number of passengers). The web browser then uses this entered information to assemble a request that is sent to a form handler. The form handler is a component that serves as an interface for the web application. This form handler could be responsible for queries submitted via multiple different forms, such as forms for round-trip flights or one-way flights.

Proper understanding of the interface exposed by the form handler can help engineers generate test cases and oracles relevant to the underlying web applications. Such an understanding may also be useful for directing maintenance tasks such as re-factoring the web pages. Finally, as we shall show, information that helps engineers comprehend web application interfaces may also help them detect anomalies in those interfaces and the underlying applications.

An understanding of web application interfaces can also be valuable for third party developers attempting to incorporate the rendered data as a part of a web service (e.g. a site that aggregates flight pricing information from multiple sources). Although web applications that are commonly used by clients may provide interface descriptions (e.g. commercial sites offering web services often offer

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 260–275, 2007.

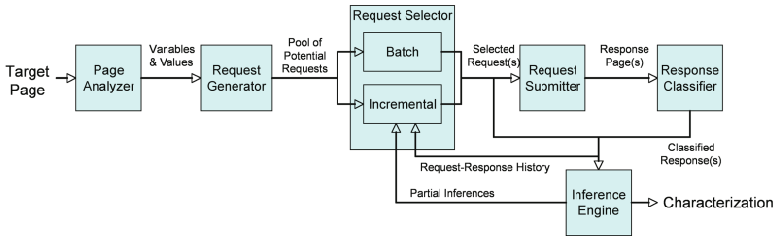


Fig. 1. WebAppSleuth architecture

a WSDL-type [1] description), many sites do not currently provide such support mechanisms. Moreover, as we shall show, the level of interface understanding that could be useful for comprehension and anomaly detection goes beyond that usually provided by such interface descriptions, and could serve as a complement to a WSDL description.

To support the various activities of both the engineers of sites, and third party developers incorporating information from other sites, we have been researching methods for automatically characterizing the properties of and relationships between variables and values in web application interfaces. In this paper, we present a methodology for characterizing the interface of a web application. Our methodology involves making requests to a target web application, and analyzing the application's responses to draw inferences about the variables and values that can be included in a request and the relationships among those variables and values. We identify three specific types of inferences, all of which have the ability to find anomalous behavior in and help increase understanding of web applications. To enhance the scalability of the approach, we provide a family of techniques for selecting requests to submit to the application.

We report the results of empirical studies of this approach, in which we apply it to six non-trivial, real-world web applications from various sources (academic, government, and commercial). Our studies show that our inferences are useful for finding anomalous behavior in these applications. In addition, we show that for these applications, our request selection techniques can reduce the number of requests needed to find correct inferences and filter out incorrect inferences, enhancing the scalability of the approach.

2 Methodology

Our methodology works by selectively submitting requests to a web application, and using the responses to those requests to discover relationships between variables and values in the application. Figure 1 shows the overall architecture for our web application interface characterization methodology, WebAppSleuth, with various processes (sub-systems) in the methodology shown as boxes. WebAppSleuth begins with a *Page Analyzer* process, which statically analyzes a target page containing a form generated by the web application. The *Page*

Analyzer identifies all variables associated with the fields in the form, and then associates a list of potential values with each identified variable. For each pull-down, radio-button, or check-box variable, the *Page Analyzer* obtains values from the possible values defined in the form. For text-type variables, the *Page Analyzer* prompts the user to supply values that may elicit a correct response from the web application.

Next, the *Request Generator* creates a pool of potential requests by exploring all combinations of values provided for each variable, as well as cases where variables are missing. Given this pool of requests, the *Request Selector* determines which request or requests will be submitted to the target application. There are two general request selection modes: *Batch* (requests are selected all at once) and *Incremental* (requests are selected one at a time guided by a feedback mechanism). The *Request Submitter* assembles the http request, sends it to the web server, and stores the response. This response is classified by the *Response Classifier*. The selected request and the classified response are then fed into the *Inference Engine*, which infers various properties about the variables and values used in submitted requests.

Currently, our methodology analyzes a single form handler within a web application. The form handler is assumed to be stateless and deterministic with respect to its inputs. Numerous important web applications satisfy (or mostly satisfy) these requirements, including applications that support travel reservation searches (e.g. Expedia), mapping applications (e.g. MapQuest), product searches (e.g. BuyAToyota's used car search) and other search sites (e.g. NSF's funding search). For sites that do not fully satisfy these requirements, it is often possible to approximate them by temporarily controlling the state (for developers characterizing their own site) or by limiting the time frame within which the site is accessed to limit potential changes to the underlying state.

In the following sections, we explain (1) how we classify responses, (2) how we use requests and responses to generate inferences, and (3) how we select the requests we are submitting, including methods that use previously submitted requests with responses and generated inferences to guide the request selection process.

2.1 Classifying Responses

After submitting each request, we classify the response. The user must choose between one of two methods for classification depending on the types of responses received and the types of inferences they wish to make. For some sites, the response is either some piece of information (i.e. a map for MapQuest) or an error message. Therefore our first method is to classify responses as either "Valid" (returns the request information) or "Invalid" (returns an error message). To classify these types of responses, our methodology searches for substrings in the result that match simple regular expressions.

Our second method is to extract a set of results from the response (for the inference algorithms that require valid/invalid classification, an empty set is invalid, and any non-empty set is valid). For example, for BuyAToyota the response

page includes a set of identifiers representing cars, possibly with links to additional pages with more cars. We collect this set of identifiers (iterating through the additional pages if necessary), and store these as the classification for the request.

2.2 Discovering Inferences

We have devised a family of inference algorithms to characterize the variables that are part of a web application interface, and the relationships between them. The algorithms operate on the list of variable-value pairs that are part of each submitted request, and on the classified responses (valid/invalid or a set of returned results) to those requests.

To facilitate the explanation of the subsequent algorithms we use examples that are further explored in our study in Section 3. Also, we simplify the terminology by defining a *valid request* as one that generates a valid response from the application, and defining an *invalid request* as one that generates an invalid response. For space reasons, detailed algorithms and descriptions are omitted, but can be found in [2].

Variable Classes and Values. It is common for web applications to evolve, incorporating additional and more refined services in each new deployment. As an application evolves, it becomes less clear what variables are *mandatory* (required in every valid request), and what variables are *optional* (may be included or absent in a valid request). Distinguishing between these classes of variables is helpful, for example, to anyone planning to access the web application interface, and to developers of the web application who wish to confirm that changes in the application have the expected results in the interface.

In addition to assisting developers with evolving applications, we can identify anomalies in the application by finding *mandatorily absent* variables (variables absent in every valid request). There are two potential reasons mandatorily absent variables may be identified: 1) the web page or web application contains a error (e.g. a field was left in a form but is no longer used by the web application) or 2) additional requests are needed to provide an appropriate characterization of that variable.

Our algorithm identifies as mandatory any variable that appears in all valid requests and is absent in at least one invalid request. Our algorithm identifies as optional any variable that appears in at least one valid request and is absent in at least one valid request. Our algorithm identifies as mandatorily absent any variable that is absent in all valid requests and appears in at least one invalid request.

In addition to finding mandatory, optional, and mandatorily absent variables, we also find the range of values for variables that produced valid responses. This allows us to detect values that never return valid results. These values could indicate that there are problems with the web application (e.g. the form includes a value for a variable that is no longer used in the application), that more requests need to be made, or that there exists an opportunity for improving

Table 1. MapQuest Requests and Variable Implications

	address	*city*	*state*	*zip*	Implication	At least one-of
1	absent	absent	present	absent	$address \implies$	$state$
2	absent	absent	absent	present	$address \implies$	$state \vee zip$
3	present	absent	absent	present	$address \implies zip$	$state \vee zip$
4	present	present	present	absent	$address \implies zip \vee (city \wedge state)$	$state \vee zip$
5	present	present	present	present	$address \implies zip \vee (city \wedge state)$	$state \vee zip$

the web application (e.g. a possible value for a variable represents a value that does not exist in the current state of the database, and filtering the values in the form based on this state could be useful).

To find the range of values, our algorithm keeps track of the values that appear in requests (distinguishing between those that appear in valid and invalid requests) and reports a list of values that appeared in valid requests for each variable. To reduce the number of falsely reported value inferences, the algorithm reports an inference for a variable only after all values included for that variable have been used at least once.

Variable Implication. Sometimes a request that contains a particular variable can be valid only if other specific variables are present. Identifying such relationships between variables is helpful for understanding the impact of application changes, and for avoiding sending incomplete requests to the application.

To investigate this type of relationship, we began by defining the notion of implication as a conditional relationship between variables p and q, namely: if p is present, then q must be present. After examining existing implications on many sites we decided to expand our attention to implications in which the right hand side is a proposition in *disjunctive normal form* and does not contain negations or the constant TRUE. This guarantees that our implications are satisfiable but not tautological. Further, this type of implication (referred to henceforth as a "standard" implication) is relatively simple to understand because it can easily be mapped to the variables' expected behavior.

Our technique constructs an implication for each variable in the application by iterating through submitted requests, and adding clauses to the implication for requests in which the set of variables present is not a superset of the variables in any other clause in the implication. For a basic notion of how our technique operates consider Table 1, which shows the process for constructing the *address* implication for MapQuest across a sequence of requests. For the first two requests, *address* is not present in the request, so we do not update the implication. The third row includes *address* and *zip*, so we need to add the clause *zip* to the right side of the implication. The next request includes *address*, *city*, and *state*, but does not include *zip* (the only variable included in the clause in the implication so far), so we add the clause $city \wedge state$ to the implication. Finally, the fifth request includes all four variables, a superset of the variables included in either of the existing two clauses, so a new clause is not needed.

In addition to standard implications, we use a similar algorithm to detect two other types of inferences. One of these is the "at least one-of" inference. This inference is a proposition in disjunctive normal form like those found on the right side of our implications. Only one of these is created per site. The last column of Table 1 shows how an "at least one-of" inference is found for the MapQuest site.

The other type of inference is value-based implication. This inference is an implication in which the left side has the form of $p = q$, where p is a variable and q is some value for that variable. We create one of these for each value of each variable in the site.

Value Hierarchies. It is often the case that when given two values for a variable, one of them should always return a subset of the results returned for the other value. Consider the case for real estate search engines, which typically provide a "minimum price" variable. As the minimum price increases, if all other variables are held constant, the returned results should be a subset of the results for lower minimum prices. Such relationships cause a hierarchy of values to exist. In the case of minimum price, this is a simple linear hierarchy with each lower price subsuming all of the results of the higher prices.

We represent hierarchy relationships as a graph, with a node for each value, and directed edges $p \rightarrow q$ indicating that $q \subseteq p$.

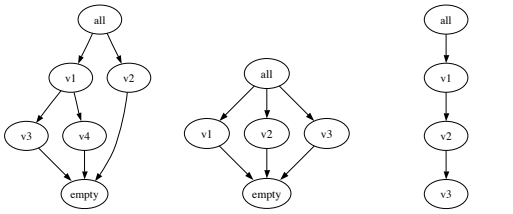
Most constrained inputs (i.e. radio button or pull-down inputs) should have a hierarchical relationship between their different values. When this is the case, the graph is a directed acyclic graph with a single root node, where that root node represents an "all" or "don't care" value for the variable (Figure 2(a)).

Anomalies in the structure of these graphs can be useful for finding problems in the web application. For example, a common anomaly seen in the applications used for our study in Section 3 is the presence of values without edges leading to them from the "all" value. This usually indicates that there were results that did not appear when the variable was set to its "all" value, but did appear under some other circumstance.

There are two special cases of the hierarchy pattern that appear often enough in web sites to warrant special consideration. The "flat" pattern (Figure 2(b)) often occurs when the underlying application looks only for exact matches of the values for the variable (excluding the "all" value).

The other special case is the "ordered" pattern (Figure 2(c)). This represents variables with values that indicate progressive restriction. The minimum price variable mentioned above is an example of this case.

Similar to our methodology for finding implications, our hierarchy inference methodology begins by creating a potential hierarchy for each variable in the application. Each potential hierarchy has two n by n boolean arrays, where n is the number of possible values for the associated variable. One of the arrays, *subset*, keeps track of whether we have found a case in which the subset relationship holds between the two values. The other array, *notSubset*, keeps track of whether we have found a case where the subset relationship does not hold



(a) Hierarchy (b) Flat (c) Ordered

Fig. 2. Example Hierarchies

between the two values. Each of these arrays is initialized with "false" in each of their cells. Then, as each request *R1* is submitted and classified, these arrays are updated.

To display hierarchical relationships, we iterate through all the possible combinations of values. If the cell in the *subset* array is "true" and the cell in the *notSubset* array is "false", we place an edge between the nodes. Beyond this there are two optimizations that can be made to make the graph more readable. The first is to combine values that return the same result into a single node (frequently we find several values that always return the empty set). The second optimization is to remove "transitive" edges from the graph. A transitive edge is any edge (u, v) where there also exist edges (u, u_1), ... (u_n, v). Currently our tool outputs the graph in dot format, which can then be read into GraphViz [3].

2.3 Selecting Requests

One of the fundamental challenges for characterizing a web application through directed requests is to control the number of requests. Larger numbers of requests imply larger amounts of time required to collect request-response data (for Expedia, one of the sites we study in Section 3, each request took about 30 seconds) and this slows down the inferencing process. In addition, our techniques are sensitive to the state of the underlying database, so when applying them to a live web application, we need to limit the time frame within which the requests are made to obtain consistent results.

To address these problems, the *Request Selector* can either select a sample of requests from the pool up-front, or it can operate incrementally by selecting a request based on previous results and continue selecting requests until the user no longer wishes to refine the inference set.

We consider two batch selection approaches. The first approach, Random, simply selects a set of random requests from the pool of requests without repetition. The second approach, Covering-Array, utilizes covering arrays [4] to determine the set of requests to submit. In general, covering arrays ensure that all n-way combinations of values are covered by the selected requests. For a given site with m variables, we consider all n, such that $1 \leq n < m$ (when $n = m$, all generated

requests are included). We used a tool developed by Cohen et al. [5] that uses simulated annealing to find covering arrays.

We consider one incremental approach, Inference-Guided, which selects requests based on requests already submitted and inferences already derived. To select which request to submit, for each unsubmitted request, this approach determines an award value, and selects the request with the highest award value. To determine an award value for each unsubmitted request R_u, we consider those requests that differ from some submitted request R_s in one variable (all other unsubmitted requests are assigned an award value of 0). We focus on this set of requests because it seems that similar requests are likely to return similar results, and we can therefore use the classification of R_s as a predictor for the classification of R_u. The award value of R_u is equal to the number of potential inferences that would be changed if R_u has the same classification as R_s.

Inference-Guided selection requires that some requests be submitted before it can begin to compute award values for other requests. We use two approaches for this. One approach begins by randomly selecting the initial requests. Another approach uses the Covering-Array tactic (for $n = 2$) to select an initial set of requests to submit, and then incrementally selects additional requests.

3 Empirical Evaluation

The goal of our study is to assess whether our methodology can effectively and efficiently characterize real web sites. In particular, we wish to answer the following research questions:

RQ1: What is the effectiveness of the characterization? We would like our characterization to be useful for understanding and finding anomalies in web applications. Therefore, we examine the inferences generated for various sites, and consider how they reflect the observed behavior of those sites.

RQ2: What is the tradeoff between effectiveness and efficiency? As the number of requests submitted to a web application increases, the quality of the inferences we can obtain should improve. However, the number of requests that can be made is limited by practical considerations. Therefore, we examine how the quality of inferences varies as requests are selected.

3.1 Objects of Analysis

Our objects of analysis (see Table 2) are six applications from various domains and implemented by various organizations. Three of them, MapQuest, Expedia, and Travelocity, have been used in other studies [6,7] and are among the top-40 performers on the web [8]. BuyAToyota is an application to search for Toyota certified used cars at local dealerships. NSF is an application supporting searches for NSF funding opportunities. UNL is a job search application maintained by the University of Nebraska - Lincoln human resources department.

Table 2. Objects of Analysis

Object	Relevant variables identified *by Page Analyzer*			Variables considered for analysis	Size of request pool
	Text Box	List Box	Check & Radio		
MapQuest	4	0	0	4	16
Expedia	4	5	2	9	49,996
Travelocity	4	7	1	9	49,996
BuyAToyota	2	5	0	5	33,408
NSF	1	7	0	7	72,576
UNL	2	4	0	4	42,345

Table 2 lists the numbers of variables identified by our *Page Analyzer* on the main page produced by each of our target web applications, at the time of this analysis, subdivided into basic input types, the numbers of those that we used for our analysis, and the total of number requests in the initial request pool for each of these applications. To keep the total number of requests manageable, we limited the variables and values for those variables that we considered as well as choosing relatively static web sites. When choosing which variables and values to consider, we attempted to select them such that an interesting, but representative range of behaviors for the web applications was explored. As we show in Section 3.4, we did not always achieve this.

3.2 Variables and Measures

Our study requires us to apply our inferencing algorithms on a collected data set of requests and responses to characterize the objects of study. Throughout the study we utilize four request selection procedures corresponding to those described in Section 2.3: Random, Covering-Array, Inference-Guided (Random), and Inference-Guided (Covering-Array).

To quantify the impact of the request selection algorithms, we compute the recall and precision as we select requests. To compute recall and precision, we had to define a set of inferences as a baseline (the "expected" inferences). For each application, we defined this set as the set of inferences reported when all requests were selected. *TotalExpectedInf* is the cardinality of this set. Then, after submitting a subset of the requests, S, we can define two additional values. The first, $ReportedExpectedInf_S$, is the number of inferences from the set of expected inferences that were reported after submitting S. The second, $ReportedInf_S$, is the total number of inferences reported after submitting S. Finally we get:

$$Recall_S = ReportedExpectedInf_S / TotalExpectedInf$$
and
$$Precision_S = ReportedExpectedInf_S / ReportedInf_S$$

Note that $Recall_S$ is 100% when the methodology reports all of the expected inferences after submitting S, and that $Precision_S$ is 100% if we report no unexpected inferences after submitting S.

3.3 Design and Setup

We applied the WebAppSleuth methodology to each of the objects of study. This involved tailoring our request submission and response classification routines as described in Section 2.1. For three of the sites (MapQuest, Travelocity, and Expedia), we used the valid/invalid classification method. For the remaining three sites we were able to collect a set of results (cars for BuyAToyota, funding opportunities for NSF, and jobs for UNL).

To expedite the exploration of several alternative request selection mechanisms and inference algorithms without making the same set of requests multiple times, we performed all the requests in the pool, and then applied the different mechanisms and algorithms to these results. This controlled for potential changes in the state of the web applications by giving a common set of response pages to operate on, while still obtaining results identical to what would have occurred had we applied the analysis to the site directly.

We performed the analysis 25 times with each type of *Request Selector* to control for the randomness factor in the request selection algorithms. For the Random and Inference-Guided selection each of these 25 runs selected one request at a time and generated inferences after each request, continuing until all the requests in the pool were selected. For Covering-Array, we selected 25 sets of requests for each level of interaction from one to one less than the number of variables in the application, and generated inferences for each of these sets of requests.

3.4 Results

We present the results in two steps, corresponding to our two research questions. First, we show and discuss the characterization provided by the methodology for

Table 3. Inferences Found for each Web Application

Website	Type	Inferences
MapQuest	**Optional**	$address$, $city$, $state$, zip
	Implications	$city \implies zip \vee state$, $address \implies zip \vee (city \wedge state)$
Expedia	**Mandatory**	$depCity$, $arrCity$, $depDate$, $retDate$, $depTime$, $retTime$
	Optional	$adults$, $seniors$, $children$
	Implications	$(adults \vee seniors)$
	Values	$children$: 1 of 4 values
Travelocity		**All inferences from Expedia**
	Implications	$(adults = 0) \implies seniors$, $(seniors = 0) \implies adults$
BuyAToyota	**Optional**	$model$, $year$, $price$, $mileage$, $distance$
	Implications	$(year = 2006) \implies model$
	Values	$model$: 13 of 28 values, $price$: 5 of 7 values
	Hierarchies	$model$: Flat, missing 3 edges from "all" value, empty values, $mileage$: Ordered, $year$: Flat, $price$: Ordered
NSF	**Mandatory**	$pubSelect$, $fundType$, $queryText$
	Optional	$month$, day, $year$, $organization$
	Values	$organization$: 46 of 48 values
	Hierarchies	$fundType$: Flat, $organization$: Missing edges from "all" value, empty values, other anomalies, $year$: Ordered
UNL	**Values**	fte: 7 of 8 values, $category$: 7 of 9 values
	Hierarchies	fte: Flat, missing 5 edges from "all" value, empty value, $category$: Flat, missing 1 edge from "all" value, empty value, $reportsTo$: Flat, $title$: Flat

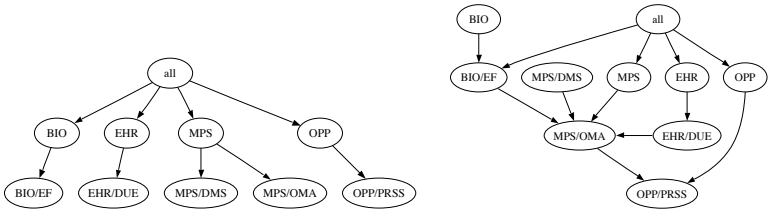
Table 4. Summary of Anomalies Found in Sites

Site and Symptom	Significance
Expedia and Travelocity: *children* had 3 invalid values	We did not consider the age variables associated with the *children* variable
Expedia: missing implications	Site returned flights in some cases when the total number of travelers was 0
BuyAToyota: missing values for *model* and *price*	We limited our search geographically, excluding results that would have filled in the missing values
BuyAToyota: $year = 2006 \implies model$	New cars were added to site as we collected requests
BuyAToyota: *model* hierarchy was flat	Models such as "Camry" did not include submodels such as "Camry Solara"
BuyAToyota: misplaced value in hierarchy	"> 100,000" miles functioned like "< 32, 767" miles
NSF: $queryText$ variable was mandatory	Blank value for $queryText$ treated different than not including $queryText$
NSF: missing values for *program*	We limited our search to active funding opportunities, excluding archived funding opportunities that would have filled in the missing values
NSF: $fundType$ hierarchy was flat	Aggregate values such as "Standard or Continuing Grant" not treated proper aggregates
NSF: *program* hierarchy had numerous anomalies	Problems with application logic
NSF: inconsistent treatment of missing variables	Design inconsistency makes maintenance more difficult
NSF: missing implication $(pubSelect = \text{"After"}) \implies day \wedge month \wedge year$	Site treated "After" the same as "Ignore" if dependent values were missing
UNL: fte and *category* had missing values	Certain values of these fields did not appear in database
UNL: *title* hierarchy was flat	Titles such as "Assistant Professor" and "Assistant Professor-Political Science" returned disjoint sets of results
UNL: missing edges in fte and *category* hierarchies	Either problems with application logic or the database state changed as we submitted requests

each target web application when the entire pool of requests is utilized. Second, we analyze how the characterization progresses as the requests are submitted and analyzed, utilizing four different request selection mechanisms.

RQ1: Effectiveness of the Characterization. Table 3 presents the inferences derived from the requests we made and the responses provided by each of the target applications. Overall, we were able to find anomalies on five of the six web applications, suggesting that our methodology can be used to help improve the dependability or usability of web applications. Table 4 summarizes all of the anomalies found. For space reasons we discuss just two of these in detail, the others are discussed in Reference [2].

The first example anomaly is for Expedia and Travelocity. On these applications we looked at sets of variables and values for which we expected to get identical results. However, there were two value-based implications found for Travelocity that did not appear in Expedia. These implications were the result of Travelocity never returning a list of flights if the total number of selected passengers was 0, while in some cases Expedia would return a list of flights. Since flight search in both of these sites is just the first step in a process for purchasing tickets and since Expedia's behavior has changed since the original set



(a) Expected (b) Actual

Fig. 3. NSF Organization Hierarchy

of requests was submitted, this difference in behavior indicates that the earlier version of Expedia probably contained a fault.

The second example anomaly was on NSF. The NSF grant search application includes a variable, organization, that allows the user to select which NSF program they are interested in. Figure 3(a) shows the expected hierarchy for an interesting subset of the values for the organization variable, while Figure 3(b) shows the hierarchy that was actually generated. The first thing to note is some missing edges (e.g. between all and BIO and between MPS and MPS/DMS). In addition, the value MPS/OMA is a child of multiple values: BIO/EF, MPS, MPS/DMS, and EHR/DUE. This occurred because particular grants could belong to multiple programs and, in this case, only one grant offered through MPS/OMA appeared in our results, and it belonged to the other programs as well. Finally, OPP/PRSS appears at the bottom as a descendant of every other node as no grants were ever returned for this value.

RQ2: Effects of Request Selection. Figure 4 presents our results with respect to the precision and recall of the Inference-Guided and Random request selection techniques, for three of the six web applications (with only 16 requests, MapQuest is too small an example for request selection to be useful, Travelocity had results nearly identical to Expedia and NSF had results similar to BuyAToyota). In each of the graphs, the x-axis represents the number of requests selected from the pool, and the y-axis represents the average recall (left column) or precision (right column) over the 25 runs. Each of the lines represents one of the request selection techniques, and the legend below indicates which line corresponds to which technique.

On two applications, Expedia and UNL, Inference-Guided request selection (with Random or Covering-Array seeds) had average recall equal to or better than Random or Covering-Array request selection regardless of the number of requests selected. On these objects we see little difference between the two Inference-Guided techniques or between the Random and Covering-Array techniques (when considering the graphs for the Covering-Array technique, the points of interest are the corners of the "steps" as these represent the collected data points, while the other points along these plots are meant to aid in their interpretation). For BuyAToyota, all the techniques were only slightly different in terms of recall throughout the process.

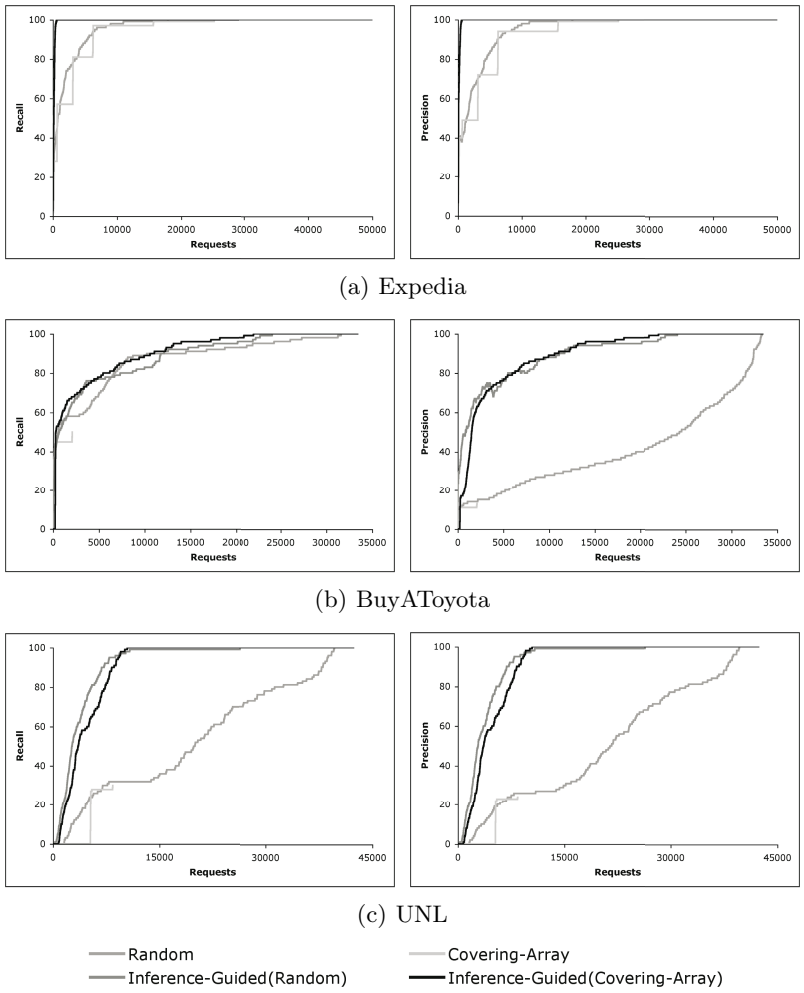


Fig. 4. Recall and precision vs percent of requests submitted

For all of the web applications, Inference-Guided request selection (with Random or Covering-Array seeds) had average precision equal to or better than Random or Covering-Array request selection throughout the request selection process. Again, there was little difference between Inference-Guided (Random) and Inference-Guided (Covering-Array) or between Random and Covering-Array.

These results are encouraging because they show that we can often dramatically reduce the number of requests required, while still reporting most correct inferences and few incorrect inferences. In particular, for the application with just valid and invalid classifications (Expedia) we needed fewer than 650 requests (1.3% of the pool) to achieve 100% recall and precision with the Inference-Guided

techniques, and 21,500 requests (43% of the pool) with Random selection. The addition of set classification and hierarchy inferences makes request selection less effective, but we can still reduce reported incorrect inferences quickly using Inference-Guided selection. In addition, it appears that using Covering-Array techniques does little overall to improve the recall and precision of reported inferences (either by itself in comparison to Random or as a seeding technique for Inference-Guided instead of using random seeding).

4 Related Work

There has been a great deal of work to help identify deficiencies in web sites, to provide information on users's access patterns, and to support testing of web applications [9,10,11,12,13]. Among these tools, our request generation approach most resembles the approach used by load testing tools, except that our goal is to generate a broad range of requests to characterize the variables in the web application interface. There are also tools that automatically populate forms by identifying known keywords and their association with a list of potential values (e.g., zipcode has a defined set of possible values, all with five characters). This approach is simple but often produces incorrect or incomplete requests, so we refrained from using it in our studies to avoid biasing the inferencing process.

Our work also relates to research efforts in the area of program characterization through dynamic analysis [14,15,16,17,18,19]. These efforts provide approaches for inferring program properties based on the analysis of program runs. These approaches, however, target more traditional programs or their byproducts (e.g., traces) while our target is web application interfaces. Targeting web applications implies that the set of properties of interest to us are different and that we are making inferences on the program interface instead of on the program internals.

Recent approaches also attempt to combine dynamic inference with input generation [20,21]. These approaches use dynamic inference techniques to classify the behavior of the program under generated inputs to determine the usefulness of these inputs for finding faults. Our approach differs in that we want to avoid executing new inputs that will not help our characterization due to the high cost of their execution and the large number of potential requests.

In our own prior work, we have made several inroads into the problems of automatically characterizing the properties of and relationships between variables in web application interfaces. In earlier work [7] we presented static approaches for analyzing HTML and JavaScript code to identify variable types, and a dynamic approach for providing simple characterizations of the values allowed for variables (e.g., a variable cannot be empty). However, deeper characterizations of web application interfaces were not obtainable through the mechanisms that we considered. More recent work [6] presented our techniques for finding mandatory, optional and valid value and implication inferences as well as a less general version of our Inference-Guided request selection technique. This work did not

consider classification of sets of results, hierarchy inferences, or the application of covering array techniques to request selection, and looked at only three of the six applications we examined here.

5 Conclusion

We have presented and evaluated what we believe to be the first methodology for semi-automatically characterizing web application interfaces. This methodology submits requests to exercise a web application, and analyzes the responses to make inferences about the variables and values within the application interface. As part of the methodology we have introduced a family of selection mechanisms for submitting requests more efficiently. Further, the results of an empirical study of six web applications from a variety of domains indicate that the methodology can effectively derive inferences that can help with anomaly detection or understanding of the web application interface and that our Inference-Guided request selection technique can reduce the number of requests required to get correct inferences and filter out incorrect ones.

These results suggest several directions for future work. First, we would like to extend our methodology to work with different types of web applications. Second, the current non-automated steps of the methodology, customization of the request submission and response classification routines, required between four and eight hours for each of the sites we studied. Hence, we plan on leveraging patterns in web applications along with clustering techniques to build heuristic methods for automating these parts of WebAppSleuth. Finally, we will explore additional types of inferences.

Acknowledgements. Thanks to M. Cohen who provided us with her tool for generating covering arrays. K.-R. Chilakamarri participated in the early portions of this work. This work was supported in part by NSF CAREER Award 0347518, the EUSES Consortium through NSF-ITR 0325273 and the ARO through DURIP award W911NF-04-1-0104.

References

1. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language. http://www.w3.org/TR/wsdl (2001)
2. Fisher II, M., Elbaum, S., Rothermel, G.: Dynamic characterization of web application interfaces. Technical Report UNL-TR-CSE-2006-0010, University of Nebraska - Lincoln (2006)
3. GraphViz. http://www.graphviz.org/ (2006)
4. Cohen, D., Dalal, S., Fredman, M., Patton, G.: The AETG system: An approach to testing based on combinatorial design. IEEE Trans. on Softw. Eng. **23**(7) (1997) 437–444
5. Cohen, M., Colbourn, C., Gibbons, P., Mugridge, W.: Constructing test suites for interaction testing. In: Int'l Conf. on Softw. Eng. (2003) 38–48

6. Elbaum, S., Chilakamarri, K.R., Fisher II, M., Rothermel, G.: Web application characterization through directed requests. In: Int'l Workshop on Dynamic Analysis. (2006)
7. Elbaum, S., Chilakamarri, K.R., Gopal, B., Rothermel, G.: Helping end-users "engineer" dependable web applications. In: Int'l Symp. on Softw. Reliability Eng. (2005) 31–40
8. Consumer top 40 sites. http://www.keynote.com/solutions/performance_indices/consumer_index/consumer_40.html(2006)
9. Benedikt, M., Freire, J., Godefroid, P.: VeriWeb: Automatically testing dynamic web sites. In: Int'l WWW Conf. (2002)
10. Elbaum, S., Rothermel, G., Karre, S., Fisher II, M.: Leveraging user-session data to support web application testing. IEEE Trans. on Softw. Eng. (2005) 187–201
11. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: Int'l Conf. on Softw. Eng. (2001) 25–34
12. Software QA and Testing Resource Center: Web Test Tools. http://www.softwareqatest.com/qatweb1.html (2006)
13. Tilley, S., Shihong, H.: Evaluating the reverse engineering capabilities of web tools for understanding site content and structure: A case study. In: Int'l Conf. on Softw. Eng. (2001) 514–523
14. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Symp. on Principles of Prog. Lang. (2002) 4–16
15. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Int'l Conf. on Softw. Eng. (1999) 213–224
16. Hangal, S., Lam, M.: Tracking down software bugs using automatic anomaly detection. In: Int'l Conf. on Softw. Eng. (2002) 291–301
17. Henkel, J., Diwan, A.: Discovering algebraic specifications from Java classes. In: Eur. Conf. on OO Prog. (2003) 431–456
18. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Comp. Systems **15**(4) (1997) 391–411
19. Yang, J., Evans, D.: Dynamically inferring temporal properties. In: Workshop on Prog. Analysis for Softw. Tools and Eng. (2004) 23–28
20. Pacheco, C., Ernst, M.: Eclat: Automatic generation and classification of test inputs. In: Eur. Conf. on OO Prog. (2005) 504–527
21. Xie, T., Notkin, D.: Tool-assisted unit test selection based on operational violations. In: Int'l Conf. on Auto. Softw. Eng. (2003) 40–48