

Web Application Characterization through Directed Requests

Sebastian Elbaum, Kalyan-Ram Chilakamarri, Marc Fisher II, Gregg Rothermel
Computer Science and Engineering Department
University of Nebraska-Lincoln
{elbaum,chilaka,mfisher,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 must be able to understand and characterize their interfaces. Third-party programmers (professional or end user) wishing to incorporate the data provided by such services into their own applications would also benefit from such characterization when the target site does not provide adequate programmatic interfaces. In this paper, therefore, we present methodologies for characterizing the interfaces to 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 three well-known, non-trivial web applications.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Experimentation, Reliability

Keywords

Dynamic analysis, web application interfaces

1. INTRODUCTION

Web applications are among the fastest growing classes of software in use today, providing a wide variety of information and services to a large range of users. Users typically interact with these applications through a web browser, which renders web pages generated by a web application. As the user navigates or submits data, new requests are sent to the web application through its interface.

Engineers who wish to enhance, test, and maintain web applications must be able to understand and characterize their interfaces,

and one way to do this is through the use of invariants that document those interfaces. For example, engineers maintaining a travel support site like Travelocity could leverage invariants that convey what variables must be included in a request to obtain a list of flights (e.g., departure location and date, return date), what variables are optional (e.g., number of children), or whether a particular variable is dependent on the value of other variables (e.g., if the number of adults in a request is 0, then there must be some seniors; if children are present, then their age must be included). Such characterizations could facilitate the engineer's understanding of the potential behavior of the web application. Further, they may also be useful for helping assess the correctness of the web application interface, and to generate test cases and oracles relevant to the application. Such characterizations may also be useful for directing maintenance tasks such as re-factoring the web pages. For example, if a certain field cannot be empty, then input validation code for that field could be duplicated on the client side using scripting languages where it can potentially prevent requests that will not return useful results, thus reducing server load and user time waiting for responses.

Characterizations of web application interfaces would also be valuable for third party developers (either professional or end-user programmers) attempting to incorporate the rendered data as a part of a web service (e.g., for resource coalitions [16]), or for users making specific queries on a web application without utilizing a browser. Although web applications that are commonly used by clients may provide interface descriptions (e.g., commercial sites offering web services often offer a WSDL-type [3] description), many sites do not currently provide such support mechanisms. In addition, at least one class of users, end user programmers, cannot be expected to learn particular protocols or APIs in order to access applications [4]. Moreover the characterizations we are pursuing go beyond those usually provided by such interface descriptions. Such characterization becomes more challenging in the presence of numerous variables and restrictions on variable values and combinations, which are relatively common for this type of application (the interface of one web application we examined had 29 variables, several of them inter-related).

For these reasons, we have been researching methods for automatically characterizing the properties of and relationships between variables in web application interfaces. Such characterizations can be obtained statically or dynamically. In earlier work [4] 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, such as those involving dependencies, were not obtainable through the mechanisms that we considered.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WODA'06, May 23, 2006, Shanghai, China.

Copyright 2006 ACM 1-59593-085-X/06/0005 ...\$5.00.

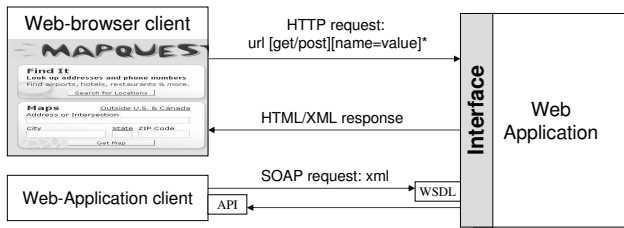


Figure 1: Web applications

In this work we address this lack, presenting a methodology for characterizing the interface of a web application by performing more sophisticated forms of dynamic analysis. Our methodology involves making directed requests to a target web application, and analyzing the application’s responses to draw inferences about the variables that can be included in a request and the relationships among those variables. We also provide mechanisms, such as a mechanism based on intelligent request selection, that enhance the scalability of the approach. Finally, we evaluate the approach’s performance on three well-known, non-trivial web applications.

The remainder of the paper is organized as follows. Section 2 provides background information on web applications. Section 3 describes our overall methodology for characterizing web applications, and also provides detailed descriptions of our inferencing and request selection techniques. Section 4 describes an empirical study exploring our methodology’s ability to characterize web applications, and the effect of our various request selection techniques. Section 5 discusses related work and approaches, and Section 6 summarizes our contribution and discusses future work.

2. BACKGROUND

Navigating through the WWW can be perceived as performing a sequence of requests to and rendering the responses from a multitude of servers. Browsers assemble such requests as the user clicks on links. Servers generate responses to address those requests, the responses are channelled through the web to the client, and then processed by the browser. Some requests may require additional infrastructure that leads to more complex applications. For example, in an e-commerce site, a request might include both a URL and data provided by the user.

Users provide data primarily through forms consisting of input fields (e.g., radio buttons, text fields) that can be manipulated by a visitor (e.g. click on a radio button, enter text in a field) to tailor a request. These input fields can be thought of as variables. Some of the variables have predefined sets of potential values (e.g., radio-buttons, list-boxes), while others are set by the user (e.g., text fields). After the client sets the values for the variables and submits the form, these are sent as request parameters known as name-value pairs (input fields’ names and their values). For example, in Figure 1 a user populates the form rendered in a browser to obtain directions from MapQuest. After receiving and interpreting the request, MapQuest provides a response (e.g., maps and directions, solicitation for more input data, error message) in the form of a markup language that is again rendered by the browser, and the cycle starts again.

As shown in Figure 1, web applications can also operate in association with other applications through direct data exchanges. For example, sites providing air-travel information often query airlines’ sites, exchanging formatted data in the process. Such interactions often occur through programmatic interfaces that have more formal descriptions. For example, the Web Services Description Language

(WSDL) [3] and the Really Simple Syndication (RSS) [11], are two popular ways to describe the interfaces between a service provider and the clients invoking the service.

As stated in the introduction, the focus of our research is on the characterization of web application interfaces. Such characterizations will be beneficial when other types of descriptions are not available (e.g., third party developers building on existing web sites without WSDL), are not appropriate (e.g., end user programmers cannot deal with complex APIs), or are not sufficient or are evolving (e.g., developers of a growing and fast changing application).

3. METHODOLOGY

Currently, our methodology analyzes the interface of a single form handler of a web application. The form handler is assumed to be stateless and deterministic with respect to its inputs. Examples that satisfy these requirements are travel reservation searches, real estate listings, and mapping applications.

Figure 2 shows the overall architecture for our web application interface characterization methodology, WebAppSleuth, with various processes (sub-systems) in the methodology shown as boxes. The methodology begins with a *Page Analyzer* process, which statically analyzes a target page 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 a list of values that may elicit a correct response from the web application. In addition, we also consider the null value to indicate that a variable is not a part of the request.

Next, the *Request Generator* creates a pool of potential requests by exploring all combinations of values provided for each variable. 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* properly assembles the http request and sends it to the target application. The web application response is stored and classified as valid or invalid 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 the relationships between variables.

The following sections provide details on the two most novel components of this architecture: the *Inference Engine* and the *Request Selector*. Further implementation details on the other components are provided in Section 4.

3.1 The Inference Engine

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 or invalid) to those requests.

To facilitate the explanation of the subsequent algorithms we utilize examples that are further elaborated on in our study in Section 4. Also, we simplify terminology by defining a *valid request* as one that will generate a valid response from the web application, that is, a response that meets the user’s expectation regarding the application behavior. We also define an *invalid request* as one that will generate an invalid response.

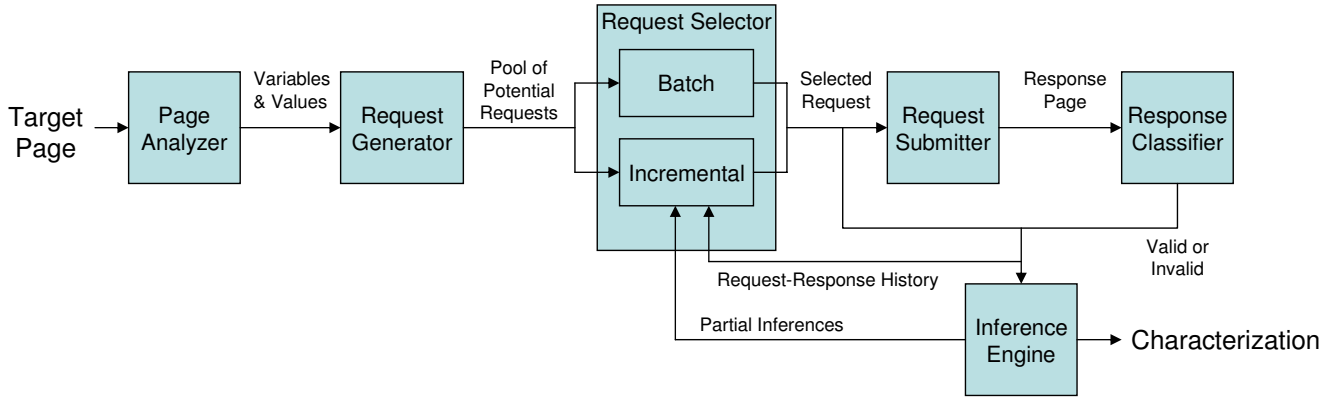


Figure 2: WebAppSleuth architecture.

3.1.1 Mandatory, Optional, and Mandatorily Absent Variables

It is common for web applications to evolve, adding additional and more refined services to each new deployment. As an application evolves, it becomes less clear what variables are required by that application, and what variables can be included in a request without being required. Distinguishing between these types 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.

We define a mandatory variable as a variable that must be in any valid request. An optional variable is one that may be included in a valid request, but is not required.

Although an interface variable should either be mandatory or optional, our inferences also identify a third type of variable that we call mandatorily absent. We define a mandatorily absent variable as one that should never be in a valid request. Finding a mandatorily absent variable implies the presence of an anomaly, since it is reasonable to assume that a variable present in a form should be used in a valid request under some circumstances. There are two potential reasons mandatorily absent variables may be identified: 1) the web page or web application contains a possible error (e.g., a field was left in a form but is no longer used by the web application), and 2) additional directed requests are needed for the methodology 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.

Observe that a variable identified as optional by the algorithm is optional in the web application interface. However, optional variables may be temporarily identified as mandatory until a valid request without that variable is submitted.

3.1.2 Variable Implication

Sometimes the presence of a variable requires other variables to be present in order to construct a valid request. Identifying such relationships is useful for understanding the impact of application changes on such dependencies, or to avoid 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 vari-

ables p and q as: 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 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 is relatively simple to understand because it can easily be mapped to the expected variables' behavior.

Our algorithm focuses on the implications between optional variables (implications involving mandatory variables would be of little value because they would just be added to the right side of every implication). The algorithm begins by initializing a set, *Implications*, with one implication, $V \implies \text{FALSE}$ for each optional variable V . Then, it iterates through all valid requests, extending each implication with an additional clause (an and'ing of all optional variables in the request) every time the implication is not satisfied by a request. Note that to generate the most general implications, the iterations through the requests progress from those with the fewest variables to those with the most variables.

To illustrate how the algorithm works, consider the set of valid requests to MapQuest shown in Table 1, and the inferred implications in the seventh column. MapQuest offers several fields including an address, city, state, and zipcode, each of them optional. For each optional variable V , the starting implication is $V \implies \text{FALSE}$ (to keep the table content simple we consider only implications with *address* on the left-hand side). The first and second requests in the table do not include variable *address*, therefore the implication $\text{address} \implies \text{FALSE}$ is satisfied, and nothing needs to be changed. The third request in the table includes *address*, therefore $\text{address} \implies \text{FALSE}$ is not satisfied, and the implication is updated by adding another clause and'ing all of the other optional variables that are present in the request, in this case *zip*. For request 4, the implication $\text{address} \implies \text{zip}$ is false, and needs to be updated by adding the clause $\text{city} \wedge \text{state}$. For request 5, the implication is satisfied and no further updating is necessary. The algorithm ends up reporting that including a street address requires the user to include either a zip code or a city and state in order for the request to generate a valid response. Note that if we had discovered a request in which *address* was the only optional variable present, this would have caused the *address* implication to be removed from the set of implications.

Another type of useful inference that can be obtained through the same algorithm is "at least one of". This is a special case of implication of the form $\text{TRUE} \implies \dots$, and can be generated using the same method used for implication. The eighth column of Table

Request	<i>address</i>	<i>city</i>	<i>state</i>	<i>zip</i>	Response	Implication	At least one-of
1	absent	absent	present	absent	Valid	$address \implies \text{FALSE}$	$\text{TRUE} \implies state$
2	absent	absent	absent	present	Valid	$address \implies \text{FALSE}$	$\text{TRUE} \implies state \vee zip$
3	present	absent	absent	present	Valid	$address \implies zip$	$\text{TRUE} \implies state \vee zip$
4	present	present	present	absent	Valid	$address \implies zip \vee (city \wedge state)$	$\text{TRUE} \implies state \vee zip$
5	present	present	present	present	Valid	$address \implies zip \vee (city \wedge state)$	$\text{TRUE} \implies state \vee zip$

Table 1: MapQuest Requests and Variable Implications

1 provides an example in MapQuest where either state or zipcode must be selected in order for a request to be valid.

3.1.3 Value-based Extensions

The previous algorithms have focused on inferences related to the presence or absence of variables, with no attention paid to variable values. Just as the characterization of presence or absence of variables could help maintainers and developers of web applications, so could characterization involving values.

For example, if no requests involving a text variable with a user-provided value generate valid responses, then additional suitable values may be required for a proper characterization. Considering values may also be useful for finding faults associated with variables whose values have been predefined through pull-down, radio-button or checkbox fields. For example, if one field has a value that always produces an invalid request, there is likely a fault in the form (a value in the form that should not be there) or the web application (failure to consider a possible value from the form).

We extended our technique for finding mandatory and optional variables to find a range of values for variables that produced valid responses. To do this, our algorithm keeps track of the values that appear in requests (distinguishing between those that appear in valid or invalid requests). It then reports a list of values that appeared in valid requests for each variable. To reduce the number of falsely reported value-based inferences, this algorithm reports an inference for a variable only after all possible values (values included in the request pool) for that variable have been used at least once. The objective is to observe enough values for a variable before determining what values constitute its valid range.

Similarly, we extended the implication algorithm. Our extension simply alters the initial set of implications to include implications of the form $(V = a) \implies \text{FALSE}$ for each variable V and possible value a . Our approach is motivated in part by the frequency with which web pages use radio buttons to determine which other fields might be required in a request. For example, payment forms often have radio buttons to select different payment types, and these payment types have different dependent variables (e.g., card number). We intend to discover this type of implication.

3.2 The Request Selector

As mentioned earlier, 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 objects of our studies in Section 4, each request took about 30 seconds) and this slows down the inferencing process. In addition, sites may not be amenable to responding to a large number of requests (for Expedia we received a warning email stating that they suspected we were launching a denial of service attack against their web site).

To address this problem, 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 is satisfied or no longer

wishes to continue refining the inference set. We have devised two request selection approaches. The first approach, *Random*, is a batch approach that simply selects a set of random requests from the pool of requests without repetition.

The second approach, *Inference-Guided*, is incremental, selecting requests based on the requests already submitted and the inferences already derived. To select which request to submit, for each unsubmitted request we determine an award value, and select the request with the highest award value (randomly breaking ties). When determining an award value for each unsubmitted request R_u , we only 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 choose to focus on this set of requests because it seems likely 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 (inferences that we have neither proven or disproven) that would be changed if R_u has the same classification as R_s .

4. EMPIRICAL EVALUATION

The goal of our study is to assess whether the proposed 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 include all the potential valid inferences (of the types specified by the algorithms in Section 3.1) that can be extracted from the responses collected from a web application. We would also like the characterization to include just the inferences that truly characterize a web application.

RQ2: What is the tradeoff between effectiveness and efficiency? Our inferencing algorithms are conservative in that they will not discard an inference unless there is data to reject it (we will validate this conjecture by addressing RQ1). This conservative approach may result in false inferences being reported when only a subset of the requests are submitted and analyzed. A limited request data set may also hinder the inferences we can derive. We wish to explore the effect of request *selection* strategies, aimed at increasing efficiency, on the methodology’s effectiveness.

4.1 Objects of Analysis

Our objects of analysis (see Table 2) are three popular applications we utilized in previous studies [4] and that are all among the top-40 performers on the web [10]. Expedia and Travelocity are travel booking applications and MapQuest is used for map lookup.

Table 2 lists the numbers of variables identified by the *Page Analyzer* on the main page produced by each of our target web applications, at the time of this analysis, and the numbers of those that we used for our analysis. To simulate multiple runs of our request selection techniques, we had to collect all of the requests in the pool identified by the request generator. Therefore, for Expedia and Travelocity, we considered only nine of their variables (Expedia had an additional 2 variables and Travelocity had an additional

Object	Relevant variables identified by Page Analyzer			Variables considered for analysis
	Text Box	List Box	Check & Radio	
Expedia	4	5	2	9
MapQuest	4	0	0	4
Travelocity	4	7	1	9

Table 2: Objects of Study.

Object	Request Pool Size	Criteria for Valid And Invalid
Expedia	49996	Valid: Available flights are displayed Invalid: More information is requested
MapQuest	16	Valid: Map returned Invalid: No map was returned
Travelocity	49996	Valid: Available flights are displayed Invalid: More information is requested

Table 3: Request Pool Size and Classification Criteria.

3, considering all of them would have required many millions of requests) in order to reduce the number of submitted requests.

4.2 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 two request selection procedures, Random and Inference-Guided.

To quantify effectiveness we compute the recall and precision of the characterization generated by the inferencing algorithms on the objects of study. A recall percentage of 100% indicates that all true inferences that characterize an application were reported by the algorithms (this might include false positives). A precision of 100% indicates that all reported inferences are indeed valid (no false positives). Let $ReportedInf$ be the number of inferences reported, let $ReportedExpectedInf$ be the number of expected inferences reported, and let $TotalExpectedInf$ be the total number of expected inferences derivable from the pool of requests, we define recall and precision as follows:

$$Recall = ReportedExpectedInf / TotalExpectedInf;$$

$$Precision = ReportedExpectedInf / ReportedInf;$$

We define the expected inferences as the set of inferences, of the types specified in Section 3.1, that are derived when the complete pool of requests is submitted. We chose to do this because we are interested in studying the effects of request selection, without regard to the quality of the pool of requests, on recall and precision.

4.3 Design and Setup

We applied the WebAppSleuth methodology to each of the objects of study. Three particular steps in this process require additional detail.

First, the *Request Generator* utilized all available potential values for each variable (including the null value which indicates that the variable is not present in a request). We used predefined values when possible. For example, for Expedia, we used the values associated with the drop-down box to select the number of “Adults” traveling. For the variables associated with *text* type fields that have no predefined values, we provided a set of potential values that can be involved in a request that would generate a valid response. For example, for Expedia we provided values for “departing from”

Website	Inferences
Expedia	Mandatory Variables: $depCity, arrCity, depDate, retDate, depTime, retTime$ Optional Variables: $adults, seniors, children$ At Least One Of: $(adults \vee seniors)$ Values: $children \in \{0\}$
MapQuest	Optional Variables: $address, city, state, zip$ Implications: $city \implies zip \vee state$ $address \implies zip \vee (city \wedge state)$
Travelocity	All inferences from Expedia Value Based Implications: $(adults = 0) \implies seniors$ $(seniors = 0) \implies adults$

Table 4: Inferences Found for each Web Application

and “going to”. The second column of Table 3 lists the generated pool size for each of the sites.

Second, since we do not have a specification for each web site’s expected behavior, we had to create one so that the *Response Classifier* could determine whether a response was valid or invalid. The third column of Table 3 lists the criteria utilized to make such a determination. Once the determination criteria were defined for a given web application, we automated the classification process by searching for the specified criteria in the returned response files.

Third, although the methodology is basically a sequential process (with a loop in case of incremental request selection), we investigated the methodology through a slightly different approach. 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 simulated the application of the different mechanisms and algorithms. We performed this simulation 100 times with each type of *Request Selector* to control for the randomness factor in the request selection algorithms.

4.4 Results

We present the results in two steps. First, we show the characterization provided by the methodology for 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 two different request selection mechanisms.

4.4.1 RQ1: Effectiveness of the Characterization

Table 4 presents the inferences derived from the requests we made and the responses provided by each of the target applications, grouped according to the types defined in Section 3.1. In Expedia and Travelocity, six variables — $depCity$ (departure city), $arrCity$ (arrival city), $depDate$ (departure date), $retDate$ (return date), $depTime$ (departure time) and $retTime$ (return time) — were identified as mandatory. Indeed, when inspecting these sites, we found that they do not provide any flight information unless those fields have been completed. Three variables were optional — $adults$, $seniors$ and $children$ — for both Expedia and Travelocity, which means that their absence does not preclude a user from obtaining a valid response from the application.

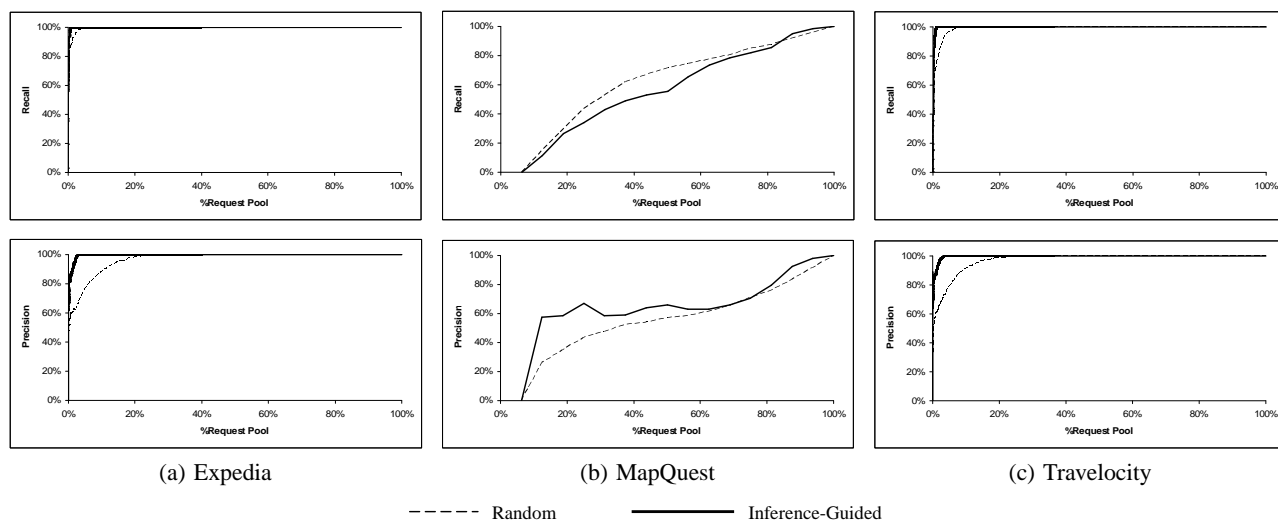


Figure 3: Recall and precision vs percent of requests submitted

Both sites also included an “at least one of” inference since either *adults* or *seniors* were present in all of the valid requests. Note that this inference is not true in practice since flight information can be obtained when the *children* variable is present and *adults* and *seniors* are absent in a request. However, the available requests in the pool were insufficient to falsify this inference (our requests including the *children* variable failed because we did not consider the age variables that are required when *children* is present). This is the same reason we obtained the inference $children \in \{0\}$. These inferences, although correct within the limitations of the pool of collected data, are an indicator that further requests are needed to provide a more accurate characterization of the site.

In spite of their similarities, we found an interesting difference between Expedia and Travelocity regarding two additional value-based implications. In Travelocity, if *adults* = 0, then the variable *seniors* must be present, and if *seniors* = 0, then the variable *adults* must be present. In practice, not having these two inferences implies that Expedia provided flight information even when no passengers were specified. Since flight finding is just the first step in Expedia’s booking process, and this behavior has been revised in Expedia since our data was collected, this inference is likely to indicate a bug in the earlier version of Expedia.

Last, MapQuest was unique in that we did not identify any mandatory variables in it. In addition, we found two interesting implications, if *city* is present we need a *zip* or *state* to locate that city, and if *address* is present, we need to know which city it is in.

4.4.2 RQ2: Effects of Request Selection

Figure 3 presents our results for each of the web applications with respect to both Inference-Guided and Random request selection techniques. In each of the graphs, the x-axis represents the percentage of requests selected from the pool, and the y-axis represents the average recall or precision over the 100 runs.

For two of the three objects of study (Expedia and Travelocity), Inference-Guided request selection had average recall equal to or better than Random request selection regardless of the number of requests selected (top graphs in Figure 3). The other object, MapQuest, was such a small example that request selection is of little help (for both Random and Inference-Guided selection it could take up to all of the requests to achieve 100% recall).

For all of the web applications, Inference-Guided request selection had average precision equal to or better than Random request

selection throughout the request selection process (bottom graphs in Figure 3). For MapQuest, with Inference-Guided request selection, the precision sometimes drops as the number of requests increases. This seems counter-intuitive, but is explainable as the result of our criteria for reporting an inference. For example, we do not report any variable as mandatory until we have at least one valid request with that variable, and one invalid request without that variable. However, this does not guarantee that the variable is mandatory, and later requests could disprove this inference.

These results are encouraging because they show that we can dramatically reduce the number of requests required, while still reporting most correct inferences and few incorrect inferences. In particular, for the two applications with approximately 50000 requests in the pool (Expedia and Travelocity) we need fewer than 2500 requests (5% of the pool) to achieve 100% recall and precision with the Inference-Guided request selection, and 18121 requests (36% of the pool) with Random request selection.

5. RELATED WORK

There has been a great deal of work to help identify deficiencies in web sites such as broken structures, bottlenecks, non-compliance with usability or accessibility guidelines, or security concerns, to provide information on users’ access patterns, and to support testing of web applications [2, 5, 6, 14, 13, 17, 18, 19]. Among these tools, our request generation approach most resembles the approach used by load testing tools, except that our goal is not to investigate the web application’s responses to extreme loads, but rather to generate a broad range of requests that help us 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 inferring process.

Our work also relates to research efforts in the area of program characterization through dynamic analysis [1, 7, 8, 9, 15, 21]. 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 [12, 20]. 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 execution and the large number of potential requests.

6. 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 variable relationships that must be considered to obtain a valid response. As part of the methodology we have introduced an inference guided mechanism for submitting requests more efficiently. Further, the results of an empirical study of three popular web applications indicate that, given a rich enough pool of requests, the methodology can effectively derive interesting inferences with an affordable number of requests.

These results suggest several directions for future work. First, further studies are needed to determine the usefulness and scalability of the methodology. To that end, we will conduct similar studies targeting a larger number of applications and building richer request pools. These will allow us to consider issues such as our methodology's sensitivity to different input values. Also, we will target web applications on which we have some degree of control such that we can assess the methodology's potential in-vivo. Such assessments will also provide insights into how best to incorporate the methodology into existing web programming and authoring environments.

Second, we will develop further support for the non-fully automated steps of the methodology. For example, we currently solicit a classification criterion to distinguish valid from invalid responses. When invalid responses are not uniquely identifiable, this task can become cumbersome and fault prone. We are exploring the use of clustering devices with which to, for example, solicit user participation only when the response cannot be automatically classified.

Finally, we will explore additional families of inferences. This exploration will consider types of inferences that are not currently present in our library (e.g., inferences involving temporal relationships), and also the application of existing inferences to other elements on the site (e.g., labels associated with the fields) and on the application (e.g., inferences on sequences of requests).

Acknowledgements

This work was supported in part by NSF CAREER Award 0347518 and the EUSES Consortium through NSF-ITR 0325273.

7. REFERENCES

- [1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, Jan. 2002.
- [2] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: automatically testing dynamic web sites. In *International WWW Conference*, May 2002.
- [3] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. Web services description language. <http://www.w3.org/TR/wsdl>.
- [4] S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel. Helping end-users “engineer” dependable web applications. In *International Symposium on Software Reliability Engineering*, Nov. 2005.
- [5] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, pages 187–201, Mar. 2005.
- [6] Empirix. E-Tester. <http://www.rswsoftware.com/products>.
- [7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, May 1999.
- [8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, May 2002.
- [9] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *European Conference on Object-Oriented Programming*, pages 431–456, July 2003.
- [10] KeyNote. Consumer top 40 sites. www.keynote.com/solutions/performance_indices/consumer_index/consumer_40.html.
- [11] M. Pilgrim. What is RSS? <http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html>.
- [12] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. In *European Conference on Object-Oriented Programming*, pages 504–527, July 2005.
- [13] Rational-Corporation. Rational testing robot. <http://www.rational.com/products/robot/>.
- [14] F. Ricca and P. Tonella. Analysis and testing of web applications. In *International Conference on Software Engineering*, pages 25–34, May 2001.
- [15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
- [16] M. Shaw. Sufficient correctness and homeostasis in open resource coalitions: How much can you trust your software system? In *International Software Architecture Workshop*, June 2000.
- [17] Software QA and Testing Resource Center. Web Test Tools. <http://www.softwareqatest.com/qatweb1.html>.
- [18] Software Research, Inc. eValid. <http://www.soft.com/eValid/>.
- [19] S. Tilley and H. Shihong. Evaluating the reverse engineering capabilities of web tools for understanding site content and structure: A case study. In *International Conference on Software Engineering*, pages 514–523, May 2001.
- [20] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *International Conference on Automated Software Engineering*, pages 40–48, Oct. 2003.
- [21] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Workshop on Program Analysis for Software Tools and Engineering*, June 2004.