Helping End-Users "Engineer" Dependable Web Applications

Sebastian Elbaum, Kalyan-Ram Chilakamarri, Bhuvana Gopal, Gregg Rothermel
Computer Science and Engineering Department,
University of Nebraska-Lincoln,
Lincoln, Nebraska, USA,
{elbaum, chilaka, bgopal, rothermel}@cse.unl.edu

Abstract

End-user programmers are increasingly relying on web authoring environments to create web applications. Although often consisting primarily of web pages, such applications are increasingly going further, harnessing the content available on the web through "programs" that query other web applications for information to drive other tasks. Unfortunately, errors can be pervasive in web applications, impacting their dependability. This paper reports the results of an exploratory study of end-user web application developers, performed with the aim of exposing prevalent classes of errors. The results suggest that end-users struggle the most with the identification and manipulation of variables when structuring requests to obtain data from other web sites. To address this problem, we present a family of techniques that help end user programmers perform this task, reducing possible sources of error. The techniques focus on simplification and characterization of the data that end-users must analyze while developing their web applications. We report the results of an empirical study in which these techniques are applied to several popular web sites. Our results reveal several potential benefits for end-users who wish to "engineer" dependable web applications.

1 Introduction

In increasing numbers, end users are using web authoring environments such as FrontPage [14], Dreamweaver [1], and Aracnophilia [12] to create increasingly sophisticated web applications. These environments allow end users to "program" web applications, working at a level of abstraction that almost isolates them from low-level scripting languages. Although often used simply to create static web pages, these environments also allow end users to create much richer applications that leverage the content available on the web, through devices such as modules that query other web applications for specific information (e.g., stock market values), templates to assist in secure remote data retrieval (e.g., https to bank services), and programmable macros to support generic web services (e.g., access to an intranet site).

Raz and Shaw [17], discussing a class of end-user authored systems referred to as *resource coalitions*, describe an example of one such application. In this example, Pat, a non-programmer with access to a web authoring environment, has a combination of chronic medical conditions (including diabetes) and an interest in health and fitness. Using a web authoring tool, Pat creates a personal web page that computes diet and exercise safety ranges. She uses this web page daily by entering her current medical status (e.g., blood sugar levels) and previous day's medication intakes. Her web page then queries other on-line resource sites that offer detailed pharmaceutical information and exercise suggestions, and displays information that helps her manage her medical conditions.

As Shaw observes [22], it is important that applications such as this be *dependable* that is, sufficiently reliable for their intended purposes and the dependability risks in scenarios such as this can be significant. For example, when Pat analyzes the input fields in the pharmaceutical sites she draws on, attempting to program the necessary queries, if she selects the wrong input field (e.g., type-1 instead of type-2 diabetes), or forgets an optional but important input field (e.g., unit measure for glucose), or uses AND to mean OR to integrate the information from two vendors, her computations may be incorrect. The potential impact of such errors is compounded when other diabetic users utilize Pat's page as an information resource.

In this context, Pat and other end-user "programmers" like her are part of a growing trend, in which end users rather than professionals are creating "software", and in numbers far exceeding those of professional programmers [21]. Further, many of these end users are interested specifically in the ability to program web applications that utilize web information sources, and in particular, applications that act as resource coalitions [19].

It is important to support the needs of these end users, and software engineering research has much to offer in the way of such support. Researchers wishing to support the dependability needs of end users, however, must attend closely to the unique characteristics of these users. End users are *not* software engineers, and have little interest in learning about software engineering methodologies; rather, they program to achieve other job- or life-related goals. The solutions that software engineering researchers might propose for engineers are not likely to work for end users without significant adaptation.

For example, in the web application domain, end users are not aware of the underlying technologies that support their web sites or applications, and cannot be expected to spend time learning markup languages. End users can be expected to be familiar, however, with web authoring environments (e.g., Frontpage) through which they create web applications, and support for dependability could be integrated into these. Similarly, although particular web sites that users may wish to draw information from are beginning to provide data in standardized formats (e.g., through XML), the types of formats and protocols varies, and end users cannot be expected to restrict their attentions to particular protocols, or learn multiple APIs to process particular content.

To begin to understand the needs of end-user web application programmers and the ways in which we might support them, we conducted an exploratory study in which we observed several non-programmers attempting to create a resource coalition. The results of this study, which we summarize in Section 3, reveal several types of errors experienced by such end users. The most frequent type of error concerns attempts to identify and manipulate variables in an external web site being used as a source of data, in order to structure a request for data from that site.

To help address this type of error, we have created a family of static and dynamic characterization techniques that simplify the data that end-user programmers must analyze while developing web applications, reducing their likelihood of making errors. To evaluate the usefulness of these techniques and the cost-benefits tradeoffs between them, we performed an empirical study in which we applied each of the techniques to each of ten popular web sites, and measured the reduction and simplification in the information content that must be processed by users when they attempt to create an application that queries that site. Our results reveal several potential benefits for end users who wish to "engineer" dependable web applications.

2 Background on Web Applications

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 channeled through the web to the client and then processed by the browser. This process generally takes the form of some markup language and protocol (most commonly HTML and HTTP).

Some requests may require additional infrastructure which 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, or enter text in a field) to tailor a request. These input fields can be thought of as variables. Some of the variables have a predefined set of potential values (e.g., radio-buttons, list-boxes), while others are exclusively set by the user (e.g., text fields). A special type of variable is "hidden", which indicates that a field is invisible to the visitor; these variables normally have pre-assigned values that are sent as part of the requests for processing. Once 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).

Leading the sent request are the HTTP protocol header and processing methods. The header contains context information such as browser agent, content type and length, and cookie information. The most commonly used methods are GET (to retrieve the data identified by the URL and name-value pairs) and POST (to return the URL of a newly created object linked to the specified URL). The body contains the name-value pairs with proper separator tokens.

After receiving and interpreting a request message, a server responds with an HTTP response message. The response message contains a status code that defines the class of response: 1xx: request received, 2xx: action was successfully received, understood, and accepted, 3xx: further action must be taken to complete the request, 4xx: request contains bad syntax or cannot be fulfilled, 5xx: server failed to fulfill a valid request.

Web applications also often 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. The end user applications we are describing fall within this category: applications that leverage other web applications as sources of information.

Client applications in this category are more likely to utilize programmatic interfaces to access available information services as illustrated in Figure 1. One of the most popular mechanisms to support data exchange between web applications is XML - Extensible Markup Language [11]. Requests in XML are often wrapped by SOAP - Simple Object Access Protocol - a messaging protocol that encodes the infor-

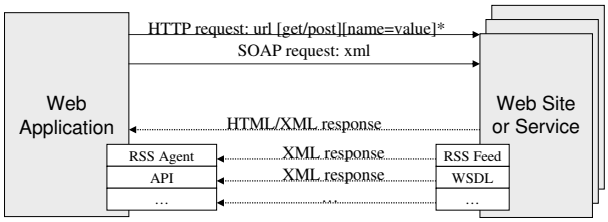

Figure 1. Web Application Service Interfacing

mation in request and response messages so that they can be handled by the HTTP protocol. XML responses may then be processed through (for example) an RSS (Really Simple Syndication [13]) or with the support of WSDL (Web Services Description Language [5]). RSS specifies how XML data-content must be structured so that it can be provided as a feed by the data producers, and clients can sign up for the feed through aggregators (programs that fetch and process the data at the client site). WSDL is an XML document that defines a communication protocol between web services and the clients invoking the web services.

As noted in the introduction, the technologies we have described here can ultimately support end-user purposes, but we cannot expect end users to learn about such a range of rapid changing technologies in order to obtain the data they need from the web, or to restrict the sources of information that they may target. Web scraping tools, which obtain information from a web page and reformat it, and web site monitoring tools, are beginning to address these challenges. Tools such as Ql2 and Connnotate [4, 16] for example, abstract the details of the underlying web technology. However, such tools often incorporate other requirements for programmers such as their own IDE, query language, and programming logic; thus, they aren't directly suitable for end user programmers. (We further discuss how to adapt such tools for end-user programmers in Section 4.1.)

3 An Exploratory Study

To obtain an initial understanding of the problems faced by end users developing web applications, we performed an observational exploratory study. The steps of this study can be summarized as follows. First, we identified a set of potential end-user web application programmers to serve as participants. Second, we ensured that these participants all had a minimum common competency in the tasks required. Third, we assigned the participants a web application development task, and then carefully recorded their activities as they attempted to complete this task. Finally, we analyzed the collected data to discover general trends, and common dependability problems worth addressing. The following subsections overview the methodology and results most relevant to this paper; complete details are available in [7].

3.1 Study Design and Implementation

As participants in our study, we selected a convenience sample of four potential end-user programmers from the secretarial staff at the Department of Computer Science and Engineering at University of Nebraska - Lincoln. These participants had experience working with the university management system, creating spreadsheets — one type of end-user program — but had no experience with professional programming languages or environments, and only one had experience with web authoring tools.

To establish a common baseline of experience among these participants, we provided a two hour tutorial acquainting them with the development of web pages, important terminology, creation of forms, and utilization of tailorable components that enable them to access data from other web sites. As a web page authoring tool we provided Arachnophilia [12] with specific tailorable components implemented as Java server scripts that the participants needed to change (in clearly flagged sections) to enable the access and data retrieval from other sites. These scripts hide the process of establishing and handling connections with the remote web site, which would likely be hidden in future development environments for end users. As part of this tutorial we also required the participants to develop a sample application that retrieves automobile prices from local dealers and compares these prices.

After the tutorial, each participant met independently with one of the co-authors, who assigned them the task of developing a web application. To complete the task, each participant used the same environment used during the tutorial, together with the tutorial documentation and an informal description of the desired application. The web application is meant to support local managers of Meals-on-wheels programs – a program organizes groups of volunteers to deliver meals to those who have difficulty obtaining them – by providing appropriate map, weather, and nutritional information from various sources to generate a schedule and provide directions for the volunteers.

We indicated to the participants that it would beneficial for the manager to automate these activities through a web application that utilizes maps, weather information, and nutritional data obtained from the web. We suggested that such an application requires the manager to provide source and destination zipcodes to retrieve the proper maps and weather information, and to provide a set of dietary restrictions to help retrieve the correct dietary data. A set of six remote web sites were provided for the participants to use as sources of information. (We purposefully constrained the number of remote sites to reduce the time participants spent in non-development tasks.) The participants were then instructed to use the input data, obtained through their web interface, to build appropriate requests for the set of preidentified sites, and perform a simple integration of the re-

Figure 2. Meals-on-wheels result screen.

sponses as presented in Figure 2. Participants were also instructed to filter out map images and advertisements. Finally, we indicated that all relevant information had to be displayed in distinct frames (nutrition, maps, weather) of the resulting page.

Once a participant understood the task description, they were asked to perform the following activities:

1. set up a data entry site using Aracnophilia with functionality by which users can input their location and meal restrictions;
2. analyze the list of provided sites to identify the required forms and variables;
3. set up requests to those sites by modifying the url, variables, and values in the provided components; and
4. integrate the collected information from all sites into a single web page.

For the remainder of this process, the co-author became *the observer*, and her role was primarily passive; she kept a log of activities that included time, things that appeared erroneous or problematic,[1] breaks taken, and questions or special requests. The observer followed the same protocol with each participant, directing them to written instructions when they had questions, and addressing their concerns only when the level of frustration appeared to potentially threaten their continued participation. Sessions were audio-taped for posterior analysis. The activity was considered complete when the participants had implemented the application to the extent outlined in its description and their application passed a small series of test scenarios.

3.2 Summarized Findings

We used the data collected in this study to create a model of the "lifecycle" by which the end-user programmers built web applications. This lifecycle model is depicted in Figure 3; nodes represent discrete activities and edges represent the transitions between those activities.

[1] We had previously established a coding scheme based on a list of potential error types in web applications to facilitate this task [7].

Once the participants were satisfied with their understanding of the informal application requirements, they began their activities by either developing the web data-entry interface or by analyzing one of the remote sites. Their next step was usually to build a request to retrieve data from that remote site. This process of analyzing a site, constructing a request, and checking the response to that request was iterative, and the participants often jumped between these activities, adjusting the request until the response was the expected one.

Once a successful response was obtained, most of the participants proceeded to refine the response by filtering the content to match the appearance of Figure 2, interleaving this activity with various checks for correctness. This process was repeated with all the requested sites, and the results were then integrated and validated throughout the process. The integration of the developed web interface and the request resulting from the "analyze remote site" and "build request" activities happened at different times and at different intervals across participants.

Note that this exploratory study was performed with the goal of discovering some of the ways in which end-user programmers of web applications work, and some of the problems they face. As an exploratory study it does not, nor is it supposed to have, the level of control required by a more formal experiment, and as such it has various limitations. In particular, it employs only a small convenience sample of participants, and these participants' behavior may have been affected by the setup and tools we provided. The observer could have also influenced the end users. The timing and error data obtained were mapped onto the activities after the study in post-mortem analyses of the collected data, which could have affected their accuracy.

Still, although some of the characteristics of the lifecycle shown in Figure 3 may have been affected by the way in which we organized the participants' task, there are several general trends that appeared consistently across the participants' activities. One clear trend was the iterative nature of the end-user web application programming activity, which involved small increments that combined analysis and implementation, followed by quick validations of results.

A second interesting finding involved the time spent in and the errors inserted during each activity. These numbers, averaged across the participants, appear as circles in Figure 3, next to their corresponding activity. Two activities, *"analyze remote site" and "build request", were clearly the most problematic (accounting for 65% of total errors) and the most time consuming (accounting for 61% of total time spent) phases of this process.* During these two phases, 64% of the errors were associated with missing variables or incorrect variable or type identification, 11% involved missing values, 9% concerned GET/POST request handling, and 7% involved incorrect use of request separators.

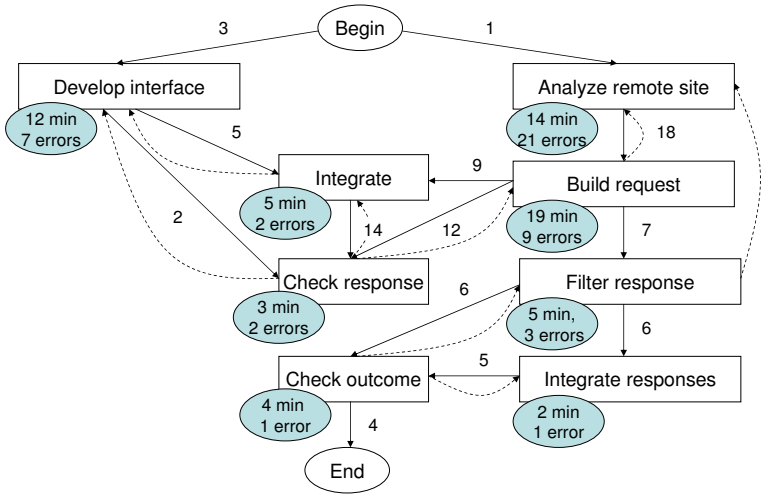

Figure 3. End-User programmer life-cycle model resulting from the exploratory study. Activities, average time allocation, and average error introduction per activity are depicted.

4 Techniques

We propose three techniques for increasing the dependability of web applications developed by end-user programmers. The targets of these techniques are the "analyze remote site" and "build request" phases of the lifecycle depicted in Figure 3. We have two reasons for focusing on these phases: 1) they are not currently assisted by the web authoring environments utilized by end users, and 2) they dominated the development cycle that we observed during our exploratory study, in terms of time spent and numbers of errors introduced.

The following subsections present each technique in turn, from simplest to most complex, first describing objectives and mechanisms, and then providing empirical data illustrating the application of the technique and its potential cost-benefits tradeoffs.

This second step, the empirical assessment of techniques, can be approached in several ways. The most comprehensive (and expensive) approach would be to integrate the techniques into an web authoring environment and assess their effects in the hands of end-user programmers. Since we have just begun to conceptualize the end-user challenges when developing web applications, and just now introduced these techniques, we chose instead to study the techniques in isolation. Doing this, we can learn about the techniques' feasibility and limitations, and their capabilities for automation, and we can identify factors that affect the techniques' performance. This information can then guide the integration of techniques and web authoring environments, and help us determine what type of user studies are worth conducting. To assess techniques we apply them to ten websites selected from a list of top-40 commercial websites [9].

4.1 T1: Revisiting Web Scraping

End users programming web applications that utilize information from other sites must currently analyze the HTML of remote sites to identify forms, their associated actions, and the variables required to build a request. We have seen that this process is costly in terms of time and errors. One of the subjects of our exploratory study expressed this by stating, "...that is too much for me to grasp.... I am confused by all this stuff....", when beginning the analysis task on mapquest.com, which has 1275 lines of code. Interestingly, only 39 lines of code in that site contain information relevant to end-user programmers, and we observed similar patterns in the other sites considered in our study.

We conjecture that the HTML analysis phase could be simplified if end users are directed toward the relevant portions of the HTML, or better yet, are isolated from the nuisances of any markup language by their web authoring tool. This conjecture is supported by similar studies in which highlighting the pieces of information most relevant to users led to savings in development time and fewer errors [15].

Mechanism and Empirical Results. Our first technique, T1, follows this notion of attempting to present an abstraction of the underlying markup language to the end user. The process begins when an end user, seeking data from a particular site, activates the end-user web scraping support engine within their web authoring tool. The engine retrieves a requested page from the site and allows the end user to drag and drop elements from the site into her design space. For example, Figure 4 illustrates the steps taken by a user to (1) navigate to Wallmart.com, (2) confirm that they are on a desired page, (3) grab a desired form (the book-search form) from that page, and (4) incorporate that form into her own space.

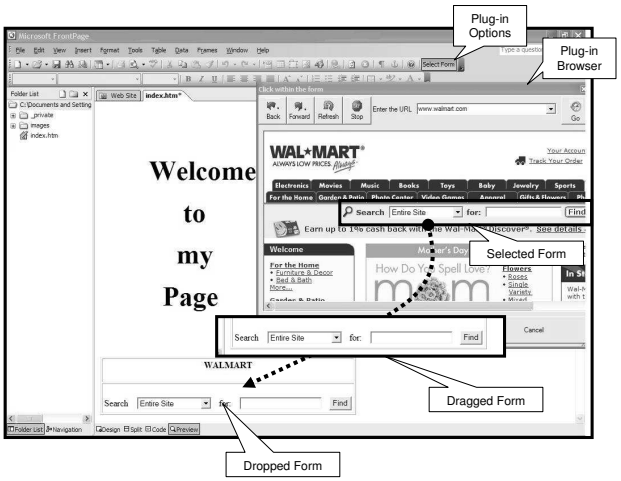

Figure 4. Scraping support within Front Page.

To evaluate T1, we implemented the end-user support engine within Microsoft Front Page as a plug-in utilizing functionality available in the .net libraries. The plug-in allows an end user to drag and drop HTML elements from a site into the end-user design space. The tool is smart enough to identify most of the relevant code in the target site that may be relevant to the form (e.g., javascript code used by a form).

We evaluated the success ratio of dragging and dropping a form from our ten target sites into the end-user design space. Table 1 summarizes the results. We found that the drag and drop was feasible for all forms on all sites. However, two of the ten forms did not function as expected when inserted in the end-user space: ebay.com had forms whose behavior was dependent on server site scripts that were not accessible, and expedia.com has a series of scripts that required full-blown browser capabilities that our prototype did not include. Overall, however, this success ratio suggests that it would be feasible to incorporate the technology available in more advanced web scraping tools into end-user web authoring tools.

Still, when items that drive the requests, such as forms, are incorporated into the end-user design space, it is up to the end-user programmer to manipulate the content corresponding to them. For example, if the user wants to set a permanent value for the zipcode because her departure point is always the same, it is still up to her to identify variable names and types to shape the proper request. This may not be trivial for some web sites. For example, one form in expedia.com uses nine fields distributed across 284 lines of code. (Table 1 presents the numbers of lines of relevant HTML code in each of the ten sites we considered.) Another limitation of the technique is that the assistance it provides for request construction is limited to establishing the URL and action form setting. The technique does not provide support to build the name-value pairs that must be part of the request.

Table 1. Quick Scraping

Website	Drag & Drop	HTML LOC	Relevant HTML. LOC	%Reduction LOC
Expedia	No	725	284	61
Travelocity	Yes	640	117	82
Mapquest	Yes	1275	39	97
Yahoo Maps	Yes	125	55	56
Amazon	Yes	1286	111	91
BarnesNoble	Yes	353	36	90
1800flowers	Yes	620	9	99
Ebay	No	511	18	96
MySimon	Yes	1062	62	94
Walmart	Yes	1944	53	97

4.2 T2: Characterization by Static Analysis

Our second technique, T2, simplifies the analysis phase by further isolating the end-user programmer from the markup language when attempting to manipulate the query to the target site. So, for example, for a dragged and dropped form the end-user programmer receives a list of characterized variables to support the manipulation. T2 further assists the "build request" phase by automatically aggregating the selected variables and values with the correct separators, taking into consideration the construction method expected by the remote site.

Mechanism. Figure 5 illustrates how T2 works. T2 adds a new layer called *Identify variable types* to the support engine that drives the prototype shown in Figure 4. After the end-user programmer selects the target form, this layer identifies the variables in that form, their types, and their predefined values.

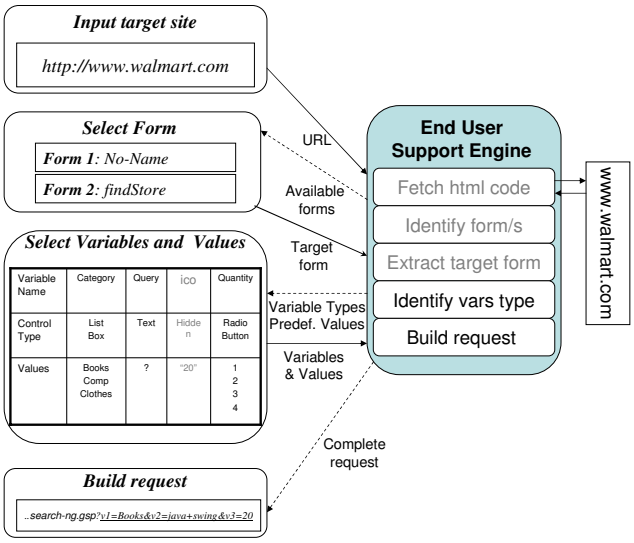

Figure 5. Characterization by static analysis.

For variables with pre-assigned values, the support engine can easily capture the list of potential values, reducing the chances that the end-user will provide unacceptable in-

put values when manipulating a request. The list of variables can be further refined by filtering out "hidden" variables that are incorporated into the request by the engine.

For variables associated with text fields that have no predefined values, the engine can check for the existence of validation scripts, and if they are present, attempt to infer field properties from them. For example, with the current prototype, we can check the six sites containing javascript to determine whether empty text fields are acceptable. (In the future we will incorporate more robust static analyses into the end-user support engine to generate more precise and powerful characterizations of text fields.)

Once the prototype is completely incorporated into a web-authoring tool (T2 current implementation operates in stand-alone mode), the end-user would select the variables from the given set as shown in Figure 5; this would help them avoid errors due to typos or names that are wrongly interpreted as variables. Also note that hidden variables are shown in gray because they are disabled for selection, which further reduces the chances of unnecessary manipulation of variables. After the variables have been selected, the end user would select input values for those variables. For variables with predefined values or text fields' inferred attributes, the engine can constrain the programmer's value choices and avoid erroneous requests due to infeasible values. After the end-user selects the variables and values, the support engine can build the complete request automatically. Based on the static analysis of the form, this process can also build the request in different formats according to the request method expected by the remote site.

Empirical results for T2. We applied T2 to our ten selected websites. Table 2 presents the results for these sites in terms of variable characterization. We can see that an end-user programmer would have to consider only a fixed list of variables, instead of having to parse tens of lines of a markup language. Furthermore, 73% of the total number of variables (hidden, listBox, checkBox, radio) have predefined values which helps control several types of errors associated with the selection of inappropriate values. We also found that six of the ten sites utilized javascript, four of them for text field validation (marked with (J) in Table 2).

The results are more appealing when hidden variables are removed from the assessment (as would occur in practice). Table 3 shows that this process leads to the average removal of 50% of the variables, with 11% deviation across sites. This leaves five of the sites with five or fewer variables for which the end-user programmer must specify values.

4.3 T3: Characterization by Dynamic Analysis

Technique T2 isolated the end-user programmer from HTML, providing instead a list of variables and, when possible, their associated predefined values. Still, when a target web site utilizes many variables that cannot be easily discarded (non-hidden variables), an end user may be faced with the challenge of selecting among these. For example, travelocity.com retains 12 variables for consideration after completion of T2. This challenge is further complicated when variable names have been poorly chosen.

Table 2. Variable Characterization with T2

Website	Total	Hidden	Text	List Box	Check Box	Radio
Expedia	29	18	4 (J)	5	1	1
Travelocity	22	10	2 (J)	9	0	1
Mapquest	9	5	4	0	0	0
Yahoo Maps	14	6	4	4	0	0
Amazon	9	4	5	0	0	0
BarnesNoble	8	3	5	0	0	0
1800flowers	3	2	1	0	0	0
Ebay	3	2	1	0	0	0
Infospace	7	4	2 (J)	1	0	0
Walmart	8	5	2 (J)	1	0	0
Total	112	59	30	20	1	2

Table 3. Variable Reduction with T2

Website	Total Variables	Reduced Variables	% Reduction
Expedia	29	11	62
Travelocity	22	12	46
Mapquest	9	4	56
Yahoo Maps	14	8	43
Amazon	9	5	45
BarnesNoble	8	5	38
1800flowers	3	1	67
Ebay	3	1	67
Infospace	7	3	57
Walmart	8	3	63
Average	12	6	50

Our third technique (T3), characterization by dynamic analysis, addresses this problem by providing a greater understanding of the potential variables involved in a request. To do this, T3 probes the target site with various requests to depict additional variable attributes that cannot be obtained through static HTML analysis.

Site probing can result in various levels of variable characterization. For example, through such probing we could learn whether there is a dependency between variables (e.g., if variable CreditCardNumber is included in a request, then variable ExpirationDate must also be included), whether there is a correlation between variables (e.g, variable From is always less than variable Until), or whether a variable is mandatory or optional (e.g., variable ZipCode is mandatory, variable Preferences is optional). We explore this last characterization in our implementation.

Mechanism. The mechanism to support dynamic characterization requires the addition of a new layer called *Infer variables attributes* to the end user support engine, as shown in Figure 6. This layer is responsible for preparing requests, sending requests to the target site, collecting the responses, and making inferences based on the responses obtained.

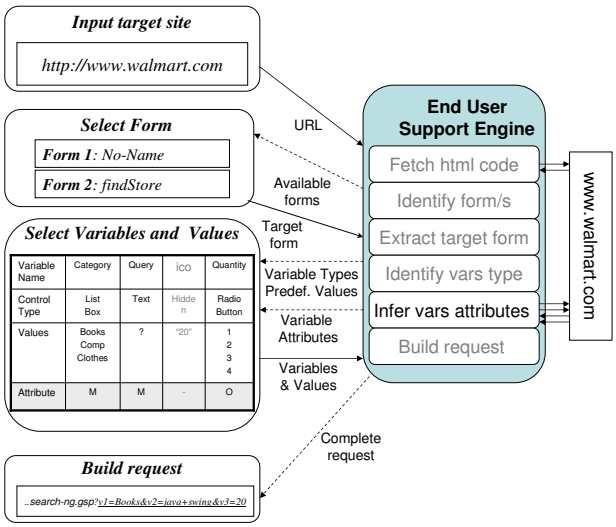

Figure 6. Characterization by dynamic analysis.

There are several aspects of this mechanism that offer a great degree of flexibility. Consider, for example, the method for combining the probing and the inferencing activities. At one extreme, the intended probing could be completed entirely prior to performing inferencing. At the other extreme, probing and inferencing could be interleaved, with the inferencing guiding the probing process. A second source of flexibility involves the degree of end user participation supported. The engine could use a brute force approach, generating many requests as soon as the target URL is provided, exploring the entire space of variable combinations, and guessing for meaningful variable values. Alternatively, the engine could begin to make requests after the end user has identified variables to assist in the value selection process. Or, the engine could request sample user inputs and use these to generate certain type of requests.

The implementation of this mechanism begins by soliciting a set of variable values for which the user expects a useful valid response. This constitutes our baseline request. Then, we systematically derive further requests by removing one or more variables at a time from the baseline request. We also derive additional requests by selecting alternative values for the variables that have pre-defined values (we avoid doing that for text-variables because it would require further end user participation limiting the degree of automation of the process). This process results in a pool of potential requests. When one of the generated requests is made and a response is collected, the implementation distinguishes an invalid response (oracle) by the presence of an incomplete or bad http status code, or by identifying keywords or page attributes provided by the end-user (e.g., "More information is needed", "No jpg map available").

There are at least two issues of concern about this implementation. First, the user's correctness expectation for the baseline request may be incorrect. As we shall see, this limits the characterization capabilities, but it does not misdirect the end user with incorrect inferences as long as the oracle is accurate. Second, our dynamic characterization implementation currently focuses on requests that work in isolation; that is, that are independent of the existence or not of previous requests that set a specific application state. Future efforts should consider sequence of requests that may, for example, set a cookie or change the user status on the remote site. Third, the process of distinguishing valid and invalid responses could involve an incorrect assessment (e.g., the keyword set provided is incomplete). Still, this process must be performed automatically since we cannot require the end user to check the outcome of tens of requests. Incorporating techniques such as clustering and outlier detection could address this challenge. Another mitigating factor is that in our studies we have observed only invalid responses to being assessed as valid, but not the opposite, which also helps us (as we shall see) ensure that our characterization is not incorrect.

As mentioned earlier, we have initially concentrated on identification of mandatory variables; these must be included in a request to generate a valid response. Algorithm 1 describes the process used to determine mandatory variables. The algorithm assumes that the target web site has been probed with the pool of generated requests and response data has been collected. If at least one request without the targeted variable generated a valid response, then that variable cannot be mandatory. Note that the two previously stated concerns limit the inferences we make about which variables are indeed mandatory, but cannot make the algorithm emit false positives (classify a variable as mandatory when it is not). The outcome of the algorithm is a list of variables that must be included in a request in order to obtain a valid response.

Algorithm 1 Inferring Mandatory Variables

1: $Requests =$ Generated pool of potential requests
2: $Mandatory =$ All Variables
3: **for all** $Request_j \in$ Requests **do**
4: **if** $Response_j$ is Valid **then**
5: **for all** $Var_i \notin Request_j$ **do**
6: Remove Var_i from $Mandatory$
7: **end for**
8: **end if**
9: **end for**

Empirical results for T3. We applied T3 to our ten websites; the results are summarized in Table 4 and include the number of requests made, the number of valid and invalid responses, the time required to make the inferences about mandatory variables, the response times from the remote site, and the number of mandatory variables detected.

Table 4 provides several interesting data points. First, through T3, mandatory variables can be recognized which can ensure that the end-user programmer includes these

Table 4. Mandatory Variables Inferred with T3

Website	Request	Valid Resp.	Infer. Time (sec)	Avg. Resp. Time(sec)	Mandatory Variables
Expedia	49997	767	28	5	6
Travelocity	49996	672	28	5	6
Mapquest	16	9	<1	2	0
Yahoo Maps	256	64	<1	2	2
Amazon	32	30	<1	1	0
BarnesNoble	32	30	<1	1	0
1800flowers	2	2	<1	1	0
Ebay	2	2	<1	1	0
Infospace	208	102	<1	2	2
Walmart	36	17	<1	1	2

variables in any request. On average, 34% of the non-hidden variables were found to be mandatory across all sites. Note also that three sites did not have any mandatory variables. For these sites, just one of several variables was needed to generate a request that would trigger a valid response. For example, in mapquest.com, the end user could choose state, city, or zipcode, and still obtain a valid response. More advanced inferencing algorithms are needed to discover that type of relationship among variables.

Second, the number of required probing requests varies substantially across sites depending on the number of variables to be considered. For example, expedia.com required 49,997 requests while ebay.com required only two requests to identify its mandatory variable. Third, a larger number of requests implied more time to infer the mandatory variables, but this was still insignificant in comparison with the response time for all required combined requests.

More surprising, however, is that while conducting the study on expedia.com and travelocity.com we discovered that our queries had the capability of affecting our context. For example, repeated queries on a site in which items are ranked can affect the overall rankings. In other cases, repeated probes can be perceived as performing a "denial of service attack", affecting target site policies. (Such experiences often forced us to collect data using several source machines, across a longer period of time).

Making T3 more scalable. As is, technique T3 seems most feasible for application to sites that have only a few variables. One approach that could be used to scale this technique up to sites with many more variables is to sample the space of potential requests. A sampling scheme would make the support engine more efficient by collecting and analyzing less data. The scheme could be adjusted to provide a fixed response time to the end user, independent of the number of variables in the target site. The cost of sampling is the potential generation of false positives when variables identified as mandatory are really optional. Alternatively, a sampling scheme could provide a level of confidence in the assessment based on the request sample size.

To investigate how a random sampling scheme might affect inferences, we studied the three sites that required the most requests to make complete dynamic characterizations, randomly selected 1%, 5%, 25%, 50%, and 75% of the request population and reran Algorithm 1. Table 5 summarizes our findings. Surprisingly, 1% of the requests were sufficient to let us identify all of the mandatory variables in expedia.com and travelocity.com. For yahoomaps.com, increasing sample size progressively reduced the number of false positives as more valid responses were included with some of the candidate mandatory variables.

5 Related Work

End users program applications in many domains. Some efforts to improve the dependability of applications transcend domains. For example, Myers et al. [10] have developed programming tools to support interrogative debugging instead of the professional code-stepping debugging approach. This effort is part of a larger attempt to reduce the long learning curve and unnatural metaphor imposed by professional debugging tools. Other efforts to improve the dependability of end-user applications are closely associated with particular domains. For example, researchers have developed a spectrum of techniques to support spreadsheet testing and debugging [2, 20].

There have also been recent efforts addressing the dependability challenges faced by web applications [6, 8, 18]. However, none of these undertakings have targeted web applications developed by end-user programmers.

Some recent work does relate to both end-user programmers and web applications. Wagner and Lieberman [23] provide a mechanism to help web site visitors to self-service and diagnose failures. The mechanism consists of an agent that provides a visualization of a web transaction history. Although this work relates to web application dependability, its target is visitors who use web applications, while we focus on end-user programmers creating web applications that interact with other web applications. Work by Shaw and Raz [17, 22] on resource coalitions includes the type of applications we are targeting. However, they focus on anomaly detection on data sources and on reconfiguration, while we focus on assisting the development of web applications that utilize other sites as sources of information. Rode and Rosson [19] report on an on-line survey of potential end-user developers of web applications. They find that 60% of end-user needs could be satisfied by tools that offer basic web data retrieval and storage functionality. The authors advocate tools that are aware of the end-user programmer's mental models, that work at high levels of abstraction. Recent work by Bolin and Miller [3] seems to be heading in that direction with the incorporation of end-user oriented programming capabilities into web browsers. Our research is partly inspired by these efforts.

Table 5. Dynamic Characterization with Sampling

Website	1 percent		5 percent		25 percent		50 percent		75 percent	
	Requests	Mandatory Variables	Requests	Mandatory Variables	Requests	Mandatory Variables	Requests	Mandatory Variables	Requests	Mandatory Variables
Expedia	500	6	2500	6	12499	6	24998	6	37497	6
Travelocity	500	6	2500	6	12499	6	24998	6	37497	6
YahooMaps	2	8	12	3	64	2	128	2	192	2

6. Conclusion

In this paper we have investigated and characterized the challenges faced by end-user programmers developing web applications who wish to leverage the web as a source of information. Our findings indicate that end-user programmers struggle with the tasks involved, especially during the analysis of the remote web site (or application) and the process of building a request to that site.

To address this problem, we have presented a family of techniques to assist end-user programmers. These techniques combine static and dynamic analyses of a target web site to simplify the analysis phase, to constrain users' choices of variables and values, and to partially automate the request building process. An assessment of these techniques over ten commercial web sites indicates that they do accomplish their objectives.

The process of understanding and assisting end users in developing dependable web applications, however, has just begun. The techniques we have prototyped must be integrated into web authoring environments, and their effect must be assessed for impacts on end-user productivity and application dependability. One approach we are investigating is the incorporation of the techniques as wizards, a common mechanism utilized by web authoring tools. We are also investigating additional techniques for dynamic analysis of remote web sites that perform deeper inferencing processes on the relationships between variables. Finally, although our work is directed at end-user programmers, it seems likely that aspects of the work could also simplify tasks faced by professional programmers.

Acknowledgments

This work was supported in part by NSF CAREER Award 0347518 and by the EUSES Consortium through NSF-ITR 0325273. We are grateful to the staff of the CSE Department at UNL who participated in our exploratory study.

References

[1] Macromedia. Dreamweaver. www.macromedia.com.

[2] T. Antoniu, P. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Int'l. Conf. Softw. Eng.*, May 2004.

[3] M. Bolin and R. Miller. Naming page elements in end-user web automation. In *Wshop. End User Softw. Eng.*, May 2005.

[4] Connotate Technologies. Connotate web mining server. http://www.connotate.com.

[5] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. Web services description language. http://www.w3.org/TR/wsdl.

[6] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Int'l. Conf. Softw. Eng.*, pages 49–59, May 2003.

[7] B. Gopal and S. Elbaum. Exploratory study: Discovering how end-users program web applications. Technical Report TR-2004-0013, University of Nebraska, Lincoln, NE, 2004.

[8] P. Graunke, S. Krishnamurthi, S. Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *Eur. Symp. Prog. Lang.*, pages 122–136, 2001.

[9] KeyNote. Consumer top 40 sites. www.keynote.com/ solutions/performance_indices/consumer_index/consumer_40.html.

[10] A. Ko and B. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Conf. Human Factors Comp. Sys.*, pages 151–158, 2004.

[11] L. Quin. Extensible markup language (xml). http://www.w3.org/XML/.

[12] P. Lutus. Arachnophilia. www.arachnoid.com.

[13] M. Pilgrim. What is RSS? http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html.

[14] Microsoft Corp. Frontpage. www.office.microsoft.com.

[15] R. Miller and B. Myers. Outlier finding: Focusing user attention on possible errors. In *Symp. User Int. Softw. Tech.*, pages 81–90, 2001.

[16] Ql2. Web ql:a software tool for web mining and unstructured data management. http://www.ql2.com.

[17] O. Raz and M. Shaw. An approach to preserving sufficient correctness in open resource coalitions. In *Int'l. W. Softw. Spec. Des.*, Nov. 2000.

[18] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Int'l. Conf. Softw. Eng.*, pages 25–34, May 2001.

[19] J. Rode and M. Rosson. Programming at runtime: Requirements & paradigms for nonprogrammer web application development. In *Symp. Human Centric Comp. Lang. Env.*, pages 23–30, Oct. 2003.

[20] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng.*, pages 110–147, Jan. 2001.

[21] C. Scaffidi, M. Shaw, and B. Myers. An approach for categorizing end user programmers to guide software engineering research. In *Wshop. End User Softw. Eng.*, May 2005.

[22] M. Shaw. Sufficient correctness and homeostasis in open resource coalitions: How much can you trust your software system? In *Int'l. Softw. Arch. W.*, June 2000.

[23] E. Wagner and H. Lieberman. Supporting user hypotheses in problem diagnosis - on the web and elsewhere. In *Int'l. Conf. Intelligent User Int.*, pages 30–37, 2004.