Investigating Data Models for Automatically Generating Tests for Web Applications
DREU 2009 Final Report

Kathryn Baldwin
Computer and Information Sciences
University of Delaware
Newark, DE USA

Camille Cobb, Caroline Hopkins, Sara Sprenkle
Computer Science
Washington & Lee University
Lexington, VA

Lori Pollock
Computer and Information Sciences
University of Delaware
Newark DE 19716

Abstract

Web applications must be dependable as the number and popularity of web applications increases, and people become more dependent on them. Web applications are difficult and expensive to test because of the large input space and frequent changes. Thus, their characteristics demand an efficient and effective way of automating the test case generation process. Current approaches to automatic test case generation for web applications do not attain all the goals of representing user behavior, maintaining good code coverage, and reducing the number of test cases. This research is based on Sant et al.'s user-session-based test case generation approach, which applies statistical language learning algorithms to create control and data models, where a control model represents the possible URL sequences and the data model represents the possible parameter values. Through analyzing user sessions, we identify factors that impact values in user sessions, and use these results to develop a set of data models for automatic test case generation.

1 Introduction

Web applications are becoming increasingly common, and people are becoming more and more dependent on these applications to accomplish tasks such as managing money and buying goods. It is therefore imperative that web applications work properly and consistently, which means they must be thoroughly tested; however, testing web applications is difficult and expensive.

One approach to making the testing of web applications cheaper and easier is to automate the testing process. Although this approach is promising, current automated testing methods are not efficient or accurate enough.

Previous research has examined several approaches to generating test cases for web applications including specification based testing, concolic testing, and user session based testing. Both specification based testing and concolic testing involve the use white-box-based test cases. Specification-based testing was an early approach to generating test cases for web applications. Because it focuses mainly on static pages and it is unable to handle dynamic components of modern web applications. Concolic testing involves the combination of concrete and symbolic execution to generate white-box-based test cases. These approaches advanced the state of the art; however, they failed to represent real user behavior in web applications.

A promising approach that is more representative of user behavior is user session based testing. User session based testing records actual user accesses to older versions of the application and parses them into user sessions, which are then used as test cases. While user session based testing is inexpensive and creates test cases that are representative of actual users, it generates too many test cases, many of which are redundant.

Our research focuses on maintaining benefits of user session based testing while improving upon the current limitations with redundancy of test cases and representativeness of users. Our goals are to create test cases that are 1) effective in terms of failure detection and code coverage 2) representative of users and 3) cost effective to generate. We base our approach on work by Sant et al. [14].

Sant et al. [14] have done work in user-representative automated test case generation. They proposed generating test cases using a model of user sessions that requires less space than the original user sessions. The model has two parts: a *control model* that represents a user's navigation through a web application as a sequence of URL requests and a *data model* that represents the user's parameter values associated with these requests.

Our research group proposed modularizing control models and data models and explored control models in depth in previous work. This paper focuses on data models. A challenge with creating data models is how to effectively represent user data (e.g., what and how much information to model) without requiring exorbitant amounts of space. A naiive approach is to generate random values, which requires very little space and could expose errors but is not representative of users' values. At the other extreme, it is not feasible for an automated test case generator to generate all possible combinations of values from user sessions.

To develop a data model efficient in both time and space that is representative of how users access the web application, we sought to identify factors that affect parameter values. We mined the user sessions for several types of information, based on our intuition about what we expected to be good potential predictors of values in terms of user representativeness. Based on these predictors, we created data models.

We then evaluate these data models to determine which models are best for which types of applications and compare them with capture-replay of user sessions. We must take into account both the costs of the data models and the benefits of their results. For the costs we will judge a data model by the amount of time it takes to generate test cases and the amount of space the test suite and the data model require. After looking at the costs we will want to see if the benefits will out weigh them. We will look at the results in terms of code coverage, fault detection, and the user representation. The important factor will be if these new data models actually improve upon the old ones.

The main contributions of this paper are

- potential predictors of parameter values
- scripts to mine useful information about user behavior
- analysis of user-behavior information to guide creation of data models
- set of data models
- results and conclusions from evaluating data models

2 Test Generation Process

Figure 1 shows an overview of the test-case generation process for web applications that we are focusing on. The test-case generation process begins with logs of user interactions with a web application, then builds control and data models, which generate test cases for the web application. Broadly defined, a web application is a set of web pages and components that form a system in which user input (navigation and data input) affects the system's state. Users interact with a web application using a browser, making requests over a network using HTTP. When a user's browser transmits an HTTP request to a web application, the application produces an appropriate response, typically an HTML document that the browser displays. The response can be either static, in which case the content is the same for all users, or dynamic such that its content may depend on user input or application state.

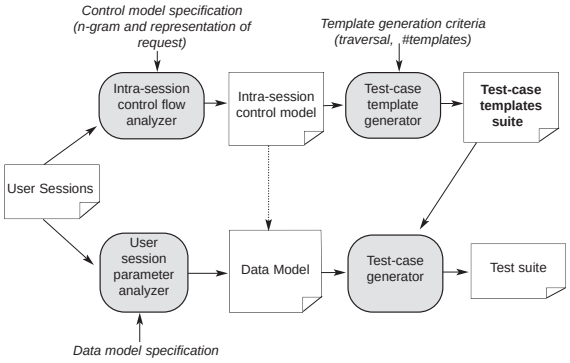

Figure 1. The Test Case Generation Process

Before the test-case generation process shown in Figure 1 begins, the user accesses are parsed and segmented to create user sessions. Each *user session* is a sequence of user requests in the form of base requests and name-value pairs. When cookies are available, we use cookies to generate user sessions. Otherwise, we say a user session begins when a request from a new Internet Protocol (IP) address arrives at the server and ends when the user leaves the web site or the session times out. We consider a 45 minute gap between two requests from a user to be equivalent to a session timing out [15].

From a set of user sessions and a control model specification, the *intra-session control flow analyzer* constructs an intra-session control model. The *test-case template analyzer* uses the control model and template criteria to produce a set of test-case templates.

Meanwhile, the user sessions are also analyzed by the user session analyzer to create an intra-session data model. From here, the test case templates and the data model are used by the test case generator to output a set of test cases—the test suite. The generator makes these test cases by assigning the values for parameters within each template, and these values are determined by the data models.

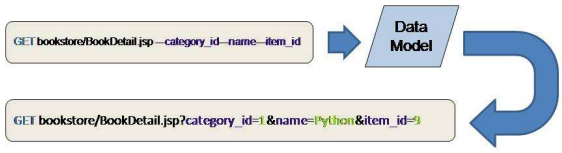

Figure 2. Process of Assigning Parameter Values to a Template

In previous work, a control model was created that looked at a URL's resource with ordered parameter names. This was determined to be a useful model because it provided more coverage information than only the URL's resource by itself. The other main contributions of previous work were that after analyzing test case *templates* (made from the control model), they proposed a practical way to use test case templates. This allows a tester to a.) more easily tune parameters to make sure the resulting test suite can meet the URL-based guarantees with lower costs. b.) reduce the size of template suites (which reduces redundancy) c.) apply multiple data models to a set of high URL+name coverage test case templates.

Figure 2 shows an example template and using a data model to assign values to the parameter values. The data models we propose in this paper use this approach.

3 Exploring Data Models

In this paper, we focus on the data models. Many different combinations of parameter values can be plugged into the same template, and the various combinations may cause the resulting request to execute different code.

3.1 Previous Work

In previous research, Sant et al. [14] developed the `Simple` and `Advanced` data models. `Simple` is based on the frequency of sets of parameter name/value combinations for the current resource, whereas `Advanced` depends on the current resource and the previous request and its "important" parameters.

Table 1 serves as an example of the probability table that `Simple`is based on. The left column contains the resource and set of parameter values found in the user sessions for a bookstore application, the middle column accounts for some of the sets of parameter values associated with the left column, and the right column contains the percentage of time the set of parameter values occured in the user sessions, given that resource and parameter names. Looking at the first row, for the resource `Bookstore/BookDetail.jsp` with the parameter names `author, category_id,` and `item_id`, `Simple`would select the set of parameter values *sprenkle, cis, 28* two-thirds of the time and values *hopkins, cis, 24* one-third of the time.

`Advanced` uses a similar table to `Simple`, but also uses the previous request and its "important parameters" as a predictor of values. Sant et al. consider "important parameters" to be parameters that remain the same across two requests.

While Sant et al.'s results were promising, there may be other information that can be used to model the data values more closely.

3.2 Our Methodology

To guide us in developing new data models, we first considered what factors may affect parameter data values. To identify these factors, we used our intuition about web applications and analyzed user sessions from four representative web application. After identifying factors, we developed data models based on these factors.

3.3 Factors That Affect Parameter Values

We analyzed the user sessions and identified three classes of factors that affect parameter values:

Parameter Interactions. We analyzed how parameters depend on each other. Sant et al. had looked at the probability of parameters as a set within one request but parameters may be related in other ways.

History. History provides context for a user's behavior. If a user has done something before how likely are they to do it again? When a user keeps going back to a certain page, does that mean the page and its parameters are more important or less? Why does a user revisit a page?

User Roles. Users often have specific roles in an application or a specific purpose when using an application. A user's intent can change, which may have an affect on the parameter values.

Given Resource and Parameter Names	Sets of Parameter Values	Probability
Bookstore/BookDetail.jsp –author–category_id–item_id	hopkins, cis, 24 sprenkle, cis, 28	33.3% 66.6%
Bookstore/BookDetails.jsp —formAction—item_id—rating	update, 24, 3 update, 28, 5,	25% 75%

Table 1. Example of `Simple`'s probability table

All of these intuitions focused us more on user behavior and how that will affect the assignment of the parameter values. From these results, we developed new data models and a user-role specific approach to generating test cases, as described in the remainder of this section.

3.4 Data Models

Based on our analysis described in the previous section, we developed two main types of data models, based on the information they use as predictors: *parameter-interaction*-based and *history*-based. These predictors can also be combined to create *hybrid* models.

3.4.1 Overview of Data Models

Our data models follow the same process as Sant et al.'s. The role of the data model in the test case generator process is to provide the probabilities used by the test case generator to assign values to the parameters. An example probability table for `Simple` is in Table 1. A probability table accounts for what factor you are looking at, the possible values, and the percent probability that those values will occur, based on the user sessions. The data model will then select from those weighted values and assign them to their corresponding parameters. Depending on the data model, different factors are accounted for and therefore different test cases are generated.

3.4.2 Parameter Interactions

Through our analysis of user sessions, we found that there are different types of parameter interactions within a user session. *Parameter interactions* are how one parameter name and value could relate to another parameter name and value. These interactions can occur within one request or multiple requests.

Sant et al's `Simple`used the set dependent *parameter interaction*. A set dependent interaction is the relationship between the entire group of parameters with a specific resource. A pro of using `Simple`is: user representativeness, (the set of values are what users actually used). Where as a con is that it is possible to have different combinations than the sets the users had. Using at other *parameter interactions* can help represent these other possibilities.

Broadly defined, the different interactions range from parameter indepedency to `Simple`'s parameter set dependency. By exploring all types of interactions we hope to ascertain what the "important" parameters are.

From there we found how often each interaction occured and if there was a pattern to the values. From these patterns we formed two hypotheses for new data models.

Since `Simple`covered one end of the spectrum, we decided to build the data model for the other end. We built the `Independent` data model to examine the indepedencies of parameters. `Independent` looks at each parameter name and its specific frequency to decide the parameter value. A pro of `Independent` are: new combinations are possible that were not in the set. A con of `Independent` is that some of the new combinations it comes up with may not make sense. For example `Independent` may assign the values of a username and a password that do not actually go together.

Now that both ends of the *parameter interaction* spectrum were formed we chose to build a model that would use an interaction from the middle of the range. The interaction we used is *param pair depedencies*. We noticed throughout user sessions that for a specific request, one parameter's value would affect another parameter's value. We called these pairings parameter couples. An example of this that enforced our idea was from the Course Project Manager (CPM) application. In CPM there were two types of users a grader and a group of students. A grader is a professor who is linked to a specific course. The grader parameter value would then determine the course parameter's value. This is an example of a coupled param object where the grader is the effector parameter and the course is the affected parameter. To create our data model based on this information we made a class that creates these coupled parameter objects with their percent value of occurance. We named this our `Coupling` data model. Since coupling is not always used by parameters we did have to inherit from the `Independent` data model. When there was no pair dependency the values were assigned based on the parameter independence.

One of the challenges behind the coupling data model was finding the strongest effector parameter. `Coupling` organized each line of the template based on the coupling strength of parameter pairs. We then assigned the first parameter a value from `Independent` and used the resource and that first parameter name and value to couple with the next parameter. The value would be chosen for the next parameter based on this. After chosing that value we would

move on to the next strongest couple and repeat the process. When we were finished we returned a list of the parameter objects.

A pro of `Coupling` was similar to `Independent` because new combinations were formed, but also more parts of the *parameter interaction* spectrum were represented. A con of `Coupling` is that it may revert back to `Independent` too often.

Along with these *parameter interactions* we would like to explore other possible relations. For example: are any parameters required? Instead of coupling can we use other subsets of parameters? We feel that if we can create data models to represent more of our spectrum we will be able to achieve maximum code coverage, while maintinging our other goals.

3.4.3 History

A second potential predictor is history because there is often a relationship between where a user has been and what they have done in a web application. By "history", we are referring to the previous requests in a user session, including the visited resources and the data associated with those requests.

We hypothesized that the previous resource would have an effect on the parameters of the current resource that the user accesses. This can provide highly useful because if we know the previous actions of a user, we can learn information about her possible behavior in the future.

Take as an example an e-commerce page, where you can shop for items online. A user may access an item's information page and from there they may go to a "Shopping Cart" page to buy the item they are interested in. Typically, buying a product using a shopping cart requires several steps. If we know that purchasing an item through "Shopping Cart" has a number of steps to it and that the step is a parameter for the page, then we can determine the step number the user is on (which is also the parameter value for "step"), using the previous resource.

If a user was previously on the item page and is now on the "Shopping Cart" page, we then know that the parameter `step` for the current page is likely to have the value "1". It is unlikely for a user to be on step "2" of the purchasing process if they just came from the item page; you cannot get to "Step 2" unless you have completed "Step 1", in which case the previous resource would instead also be the "Shopping Cart" page. In similar ways, other parameters could be determined from knowing other previous and current resources.

A data model could use minimum history (i.e., no history) or complete history (i.e., all requests within a session) or something in between the two extremes. The advantages to no history are that costs are lowered with respect to the amount of information that must be stored. However, no history would predict less-realistic values and would not be a very good representation of a user's behavior. On the other hand, looking at a complete history would ensure good representation (since every previous resource and current resource would be examined) yet would be very expensive in terms of the amount of information required. Since there are tradeoffs between the approaches, we implement a data model and will compare it with data models that use various amounts of history as predictors.

We implemented a data model that we call `Two Gram`. `Two Gram` uses slightly less history than Sant et al's `Advanced`model, using the previous resource, the current resource, and the current resource parameters. `Two Gram` uses the user sessions' frequencies of the values for the current parameters, given the current resource and the previous resource.

In the future it would be possible to make data models to examine more than simply the most-recent previous resource; perhaps instead we could include a couple of previous resources. This would be a good method to try because it would increase the amount of user representation. It might also be possible to do more work with combining both the history and the parameter interaction predictors. Perhaps by combining these two items we could learn even more information about the user's behavior.

3.4.4 Combining Data Models

After building data models from one factor: either *history* or *parameter interactions*, we felt that a step further was possible. Our idea was to use both factors to create a new data model. These would be called hybrid data models and would hopefully expand our range of results. Having a data model that is influenced by two different factors is more likely to represent users because users are influenced by multiple factors.

A hybrid model that we looked at was *consistency* of a parameter over two requests in a user session. There were two ideas behind this hypothesis. The first was that if a user is constantly inputting/using the same value it must be a more important value. The second is that if a user is trying to do something specific it would make sense for the parameter name and value to carry through. For example, if a user is searching for a specific book than the id for that book will be put in as a parameter value over and over again. Both *history* and *parameter interactions* influenced the data model.

`Consistent` takes the first request and assigns parameter values using `Independent`. For subsequent requests, `Consistent` decides if the value should be consistent by its probability tables. If it is, then the value carries through from the previous request, otherwise revert back to

`Independent`.

Consistency looks at the impact of two different factors, but it is also possible to look at the impact of two different models run together. For future research we plan to run both `Coupling` and `Consistent` together. By using multiple data models at once we are beginning to look at all sides of why a value gets assigned and how that represents the user. The more we are able to mirror the user the better our data models can be.

3.5 User-Specific Models

We observed that in the existing models by Sant et al. [14] all user sessions are treated equally; however, in many applications, there are intuitive groups of users and different ways that users use the application. In a bookstore application like Amazon.com, for example, some users sign in to buy a book, and others never sign in. Users who have signed in have different privileges than those who have not signed in. Our intuition was that because of these different access privileges and intentions for using the application, the patterns in the user values will also be different.

By partitioning the user sessions into groups, we can create models that are more tailored to these groups' behaviors. We refer to these tailored models as *user-specific models*. Beyond improving user representativeness, we also gain control over what types of test cases we generate because we can produce more test cases for the types of users that we are most concerned with or who are likely to access parts of the application that we are focused on testing. For example, on applications where users have very few options without signing in (e.g., course management applications like Sakai and BlackBoard), we could generate fewer test cases for users sessions without a login, which will help us be more effective in testing.

We have identified several ways to partition user sessions: individual user sessions, user sessions grouped by user, user sessions grouped based on access privileges, and the entire set of user sessions (the original approach). Individual user sessions are the most partitioned and the entire set of user sessions is not actually partitioned at all. We use these partitioned subsets of user sessions as the input in Figure 1. There are tradeoffs between these partitioning approaches. Using each individual user session as the input results in test cases that are very representative of the original user sessions but takes up a lot of space and time and may limit the variety of test cases. However, some degree of partitioning may be beneficial.

For example, in a course management application, some users are instructors and others are students. Members of these groups have access only to certain pages; however, there are also a significant number of shared pages (i.e., pages that both students and instructors have access to)—for example help pages and pages that do not require logging in. Furthermore, the likelihood of a particular parameter value for a given page could be significantly different for different types of users. The same kinds of differences can be seen between users as opposed to groups of users.

Grouping Users. To group user sessions into users, we searched the parameter names for each request in a session for password and then looked for the username parameter in the other parameters in the same request. The application-specific names for password and username were known and hard-coded into our script. Some sessions had no logins and some sessions had multiple logins (e.g., from a user mistyping their username); these sessions were considered as special cases.

Future work includes automatically determining the username and password parameters, but the information is not difficult for the application developer to provide. Currently, we ignore the sessions that contain multiple usernames; in the future, we could use natural-language techniques to determine if the root cause was a typo or we could split the session into a separate session for each username. For applications that do not require a password, we would have to create a different approach to group users, e.g., using the requester's IP address as a heuristic.

Grouping Users By Roles. For each application with distinct user roles, we identified the pages associated with each user role. For each specific user, we categorized the user on the type of pages they accessed. Sometimes a user accessed pages for more than one type of user. In these cases, we manually looked at the session and categorized it based on the relative number of requests to each page type.

To ensure that grouping user sessions would generate significantly different test cases from non-grouped sessions, we graphed the data about the frequency of resource accesses and parameter names and values from each identified user or user role and from the entire set of user sessions against each other. We saw that these values were very different and concluded that partitioning user sessions in this way could be useful.

Future Work includes grouping user sessions based on the intentions of the user (e.g. users on Amazon.com who are looking for a specific book versus simply browsing).

4 Evaluation Plan

We designed an evaluation plan to examine some of the remaining questions about data models and user-specific models. Specifically our plan seeks to and the following questions:

- Since code coverage is an important part of testing an application and important for fault detection, we want to find the answers to: What code coverage is achievable with different data models? How does the user-specific model influence code coverage? These conclusions will help to answer the questions: Which data model and/or user-specific model should be used for the most effective test case suites?

- It is necessary to know the costs of our different data models and user-specific model to be able to recommend their use. To find the costs we need to answer these questions: How long does it take the test case generator to run the data model? What are the sizes of test suites generated. What are the sizes of the templates and tables influenced by the user-specific model?

- One of the goals was to better represent users. To investigate the representation of users we will need to answer the questions how do our results mirror user behavior?

4.1 Variables and Measures

The independent variables are the data model (what choses the values for the parameters in the test case generation process) and the user-specific model (groups user sessions by type and therefore influences both the control model templates and the tables used by the data model). We have developed three new data models and implemented Sant et al.'s models for our system.

The dependent variables are the *effectiveness* of both the data models and the user-specific models, measured by code coverage and fault detection; the number of test suites to reach a desired code coverage level; the size of the resultant test-case templates for different user-specific models, measured by number of requests, the redundancy of the test cases; and the *cost* of the data model and the user-specific model, measured by the time required to apply the data model in the test-case generator and the amount of space required to store the data model's probability table.

4.2 Methodology

Our experiment consists of two phases: (1) generating test-case templates from the user-specific model or the control model and (2) assigning values to the paramemters, using the data models. We implemented steps (1) and (2) primarily in Python.

Generating test-case templates. For each subject application, we will generate 10 suites of test-case templates, containing 100 templates each. We will also generate templates using our user-specific model. To measure these templates we compared them to previous templates: accounting for the size, user representation and new combinations.

App	Classes	Methods	Statements	NCLOC
Masplas	9	22	441	999
Book	11	330	5347	7781
CPM	76	174	7031	8947
DSpace	274	1453	27136	49513

Table 2. Subject Application Characteristics

Assigning values to the parameters. We apply each of the data models to each set of templates multiple times to account for the non-determinism in the data models.

4.3 Subjects

In this paper, we target web applications written in Java using servlets and JSPs. The applications consist of a back-end data store, a Web server, and a client browser. Since our user-session-based testing techniques are language-independent—requiring user sessions but not source code for testing, our techniques can be easily extended to other web technologies.

We created 8 subject user-session sets from user requests to four publicly deployed applications. The applications were of varying sizes (1K-50K non-commented lines of code), technologies, and representative web application activities and usages: a conference website (Masplas); an e-commerce bookstore (Book) [7]; a course project manager (CPM); and a customized digital library (DSpace) [5]. Book is the same application used in Sant et al.'s evaluation [14]. Table 2 summarizes the applications' code characteristics.

Book was the only application for which an email was sent to local newsgroups asking for volunteer users. These user requests were also used by Sant et al. [14]. We collected accesses for each application over a long period of time: Masplas:2 months, CPM:5 academic semesters, DSpace:8 months.

We converted the user accesses into user sessions using Sprenkle et al.'s framework [15]. For CPM, we partitioned the user sessions by the semester in which they were collected to provide more test suite subjects to model and compare. Table 3 shows the characteristics of the collected user sessions, in terms of the number of user sessions, the total number of user requests, and the percent of statements covered.

5 Related Work

Approaches to automatically generating test cases for web applications can be categorized broadly into building

Subject	# User Sessions	# Requests	% Stmts Cvd
Masplas	169	1107	90%
Book	125	3564	57%
CPM1	58	1326	50%
CPM2	203	2393	67%
CPM3	105	1528	52%
CPM4	168	2240	54%
CPM5	356	4865	56%
Dspace	1800	22129	65%

Table 3. Characteristics of User Session Sets

static models of the web application from which tests are generated, creating test cases directly from user sessions, generating tests from models constructed from web logs, generating tests through static analysis of the program, and generating white-box-based tests through concolic testing.

There have been several efforts to statically model web applications for testing, which has proved difficult [1]. With the goal of providing automated data flow testing, Liu et al. [10] developed the object-oriented Web test model (WATM), which consists of multiple models to capture the different tiers of web application. Ricca and Tonella [12] developed a high-level Unified Modeling Language (UML)-based representation of a web application and described how to perform page, hyperlink, def-use, all-uses, and all-paths testing based on the data dependences computed using the model. Di Lucca et al. [11] developed a web application model and a set of tools for evaluating and automating web application testing. None of these models handles the dynamic features of modern web applications. Andrews et al. [2] proposed a system-level testing approach by modeling web applications with finite-state machines (FSMs) and using coverage criteria based on FSM test sequences. While this approach is promising, it can suffer from state space explosion.

Several different strategies for automatically constructing test cases directly from collected user sessions have been proposed [6, 15]. Sant et al.'s [14] construction of a statistical testing model from the web log is similar to Kallepalli and Tian [9]. By constructing a model from the web logs, they can use the model for different kinds of testing, including stress testing and statistical testing. Another usage-based control model that was developed for GUI testing, but is quite similar to the Sant et al. model [14], may be applicable to web testing [4].

The large number of user sessions generated in user-session-based testing can be reduced by test-suite reduction techniques [16, 13]. Our paper differs from this approach by reducing the test case templates before generating the test cases, thus reducing the time to generate test cases as well as the time to replay the test cases.

Several groups propose applying concolic testing to web application testing to generate white-box-based test cases with the goal of achieving branch or bounded path coverage [3, 17]. By combining concrete and symbolic execution and constraint solving, the system automatically and iteratively creates new input values to explore additional control flow paths through the PHP script. Artzi et al. [3] simulate user interaction by transforming the script to mimic button and menu inputs; the resulting test cases do not include tests that include browser-based inputs, or distinguish between most likely inputs to prioritize testing.

Halfond and Orso [8] developed a technique based on static analysis of individual Java servlets that automatically discovers web application interfaces (i.e., sets of named input parameters with their domain type and relevant values, which can be processed as a group by a servlet) and then generates test cases by providing data values. Similar to the concolic testing approach, this approach does not rely on user sessions to reveal application behavior. The tradeoff is that browser-based inputs and the distinction between most likely and least likely inputs are not incorporated into test case generation. This approach also does not test sequences of requests and, therefore, does not necessarily test code related to session state (with the exception of login). Halfond and Orso [8] mention that it would be interesting to combine their approach with user-session-based testing.

6 Conclusions and Future Work

In this paper, we propose various data models that each represents user data in a different way. We analyzed user sessions and developed data models with the goal of reducing the number of redundant test cases generated, while maintaining user representation. We identified three main predictors (parameter interactions, history, and user roles) that affect parameter values and developed data models based on these predictors.

In future work, we plan to consider other possible factors as potential predictors of parameter values and create new data models using these them.

References

[1] M. Alalfi, J. Cordy, and T. Dean. A survey of analysis models and methods in website verification and testing. In *ICWE'07, 7th International Conference on Web Engineering Lecture Notes in Computer Science 4607*, pages 306–311, July 2007.

[2] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3), 2005.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*, pages 261–272, New York, NY, USA, 2008. ACM.

[4] P. A. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 333–342, New York, NY, USA, 2007. ACM.

[5] DSpace Federation. `http://www.dspace.org/`, 2008.

[6] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, May 2005.

[7] Open source web applications with source code. `http://www.gotocode.com`, 2003.

[8] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154, New York, NY, USA, 2007. ACM.

[9] C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, 2001.

[10] C.-H. Liu, K. D. C., P. Hsia, and C.-T. Hsu. Structural testing of web applications. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2000.

[11] G. D. Lucca, A. Fasolino, F. Faralli, and U. Carlini. Testing web applications. In *International Conference on Software Maintenance*, 2002.

[12] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Int'l Conf. on Software Engineering (ICSE)*, 2001.

[13] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter. Applying concept analysis to user-session-based testing of web applications. *Accepted To Appear in IEEE Transactions on Software Engineering*, 2008.

[14] J. Sant, A. Souter, and L. Greenwald. An exploration of statistical models of automated test case generation. In *Proceedings of the Third International Workshop on Dynamic Analys is*, May 2005.

[15] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. A case study of automatically creating test suites from web application field data. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, Analysis, and Verification of Web Services and Applications*, New York, NY, USA, July 2006. ACM Press.

[16] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 587–596. IEEE Computer Society, September 2005.

[17] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 249–260, New York, NY, USA, 2008. ACM.