SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2012; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Configuring Effective Navigation Models and Abstract Test Cases for Web Applications by Analyzing User Behavior

Sara E. Sprenkle[1], Lori L. Pollock[2], Lucy M. Simko[1]

[1] *Department of Computer Science, Washington and Lee University, Lexington, VA USA, sprenkles@wlu.edu, simkol11@mail.wlu.edu*
[2] *Department of Computer and Information Sciences, University of Delaware, Newark, DE USA, pollock@cis.udel.edu*

SUMMARY

As web applications become more complex and are used more pervasively, testing demands are increasing without corresponding automated support. One promising approach to automatic test generation is statistical model-based testing, where logged user behavior is used to build a usage-based model of web application navigation, from which abstract test cases are generated. Executable test cases are then created by adding parameter values to the abstract test cases. Several researchers have proposed variations of this approach; however, no one has empirically examined the tradeoffs and implications of the different ways to represent user behavior in a navigation model and the characteristics of the automatically generated test cases from different models.

This paper reports on our exploratory study of automatically generated abstract test cases and the underlying usage-based navigation models constructed from over 19,000 user sessions across five publicly deployed web applications. Our results suggest how web testers can easily configure statistical model-based automatic test case generators for web applications toward generating tests closely related to user behavior or toward new navigations without using large additional test resources.

Received . . .

KEY WORDS: web application testing; statistical, usage-based navigation models

1. INTRODUCTION

Ensuring correct functionality of web applications becomes more critical and challenging as web applications are used more pervasively and grow in size and complexity. Effective runtime testing is crucial because practical static analysis of web applications is inhibited by the dynamic features of the languages, dynamically generated web pages, complex application state dependencies and concurrent user interactions.

Application navigation presents a unique testing challenge for web applications because users can circumvent the application's desired navigation constraints by using the browser's features, including the back button, location bar, bookmarks, or multiple windows. Tonella and Ricca [1] found that 47% of user sessions included an "infeasible" navigation, which they define as a navigation that does not follow any edge in their extracted model of the web application. The infeasible navigations are likely caused by browser-based navigation, such as clicking the browser's back button (which is not sent to the web application server) and following another link. Another example of a browser-based navigation is a user who bookmarked a page with restricted access after they logged in. When the user accesses the bookmark after the session times out, the web application

*Correspondence to: Sara Sprenkle. E-mail: sprenkles@wlu.edu

should respond with an access denial error or a request for a login; if not, the web application has a fault. Halle et al. [2] describe additional examples of navigation errors. Thus, it is important to test navigation constraints in addition to expected application navigation paths.

Several researchers proposed building an application *navigation model* [1, 3, 4, 5, 6] for use in testing and application understanding, among other tasks. Some researchers do not directly call them navigation models since navigation models are not the focus of their work. Strategies for automatically constructing navigation models of web applications include extracting them from source code [4], generating them using augmented web crawlers [1, 6], or generating them from the web application's usage logs [1, 3, 5]. In testing, the navigation model is used to generate *abstract test cases*, which represent URL navigation sequences. The abstract test cases are converted into *executable test cases* by adding a set of parameter values to be sent as name-value pairs in each request. Since testing all possible navigations is not practical, learning what navigation sequences users access through usage information is important. Usage information can be obtained by simply adding a logger to the web application server and requires no changes to the application code itself [7]. A study of a small web application indicates that usage-based, statistical model-based testing is a promising approach for (1) creating tests for the most (and least) likely user paths through the application, (2) creating tests that include browser-based inputs, (3) generating fewer tests than capturing user logs and replaying them directly, and (4) combining several users' paths through an application [5]. Another case study demonstrates using usage-based, statistical model-based testing for reliability estimations [1].

In this paper, we seek to answer open questions in statistical, usage-based models. For example, researchers studying statistical, usage-based models differ in their definitions of the navigation model. Furthermore, there is no empirical study of the tradeoffs and implications of the various ways to represent user behavior in a navigation model and the characteristics of the automatically generated test cases from different models. Without this information, it is not clear how best to achieve effective testing through the statistical model-based approach to testing web applications for different testing goals. Since the navigation model and generated abstract test cases are the basis for generating executable test cases, they have a large impact on the effectiveness of the resulting executable test cases.

We investigate how various configurations of a statistical, usage-based navigation model can affect the model's representation of user behavior, model size, and the generated abstract test cases. We expand upon our previous work [8] by analyzing significantly more user sessions and their resulting navigation models and abstract test cases and including several new and expanded analyses to draw stronger conclusions.

This paper makes the following contributions:

- Design and execution of an empirical study of automatically generated abstract test cases and the underlying usage-based navigation models constructed from over 19,000 user sessions from five publicly deployed web applications, spanning different technologies and domains.
- Results from the empirical study of navigation model variations that indicate that users' navigation behavior is not easily approximated without collecting usage information but relatively small user logs are needed to build a usage-based navigation model that enables generating test cases highly representative of user behavior.
- Results from studying automatically generated model-based test cases that suggest how web application testers can tune the generators toward generating tests closely representative of user behavior or toward new, likely navigations without using large additional test resources.

Section 2 provides background on web applications, navigation models in general, and usage-based statistical models, specifically. Section 3 describes our modularized statistical model-based test-case generation process. In Section 4, we discuss the open questions in configuring statistical navigation models and the resulting abstract test cases, and Section 5 presents an empirical study to answer those open questions and presents results and analyses from the study. In Section 6, we discuss related work. Finally, we summarize our results, provide guidance to testers, and discuss future work in Section 7.

DOI: 10.1002/stvr

2. BACKGROUND

2.1. Web Applications

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.

2.2. Navigation Models

Several researchers have proposed representing web applications via navigation models. Navigation models vary by how they are typically generated automatically: from static analysis of source code [4], from augmented web crawlers [1, 6] or from user access logs [1, 3, 5].

Static analysis of source code. Deng et al. [4] propose an "application model", where nodes represent URLs and edges represent URL links. The URL links, typically HTML anchors and form submissions, are extracted from the application source code. The approach is automated, but it is not clear if the *extraction* handles optional parameters, i.e., parameters that are not required for correct functioning of the code, or if the resulting model represents optional parameters. Optional parameters could result in different "logical" URLs . The authors admit that their approach does not handle dynamically generated URLs, and they allude to having only one start page for the application, which is not realistic of actual usage.

Using augmented web crawlers. Wang et al. essentially spider the application from a defined start page and use a combinatorial approach to input values into the application's forms to generate the navigation model [6]. Tonella and Ricca's navigation is similarly generated by spidering the application from a start page and inputting values from equivalence classes into the forms [1]. They augmented their navigation model with usage information, adding usage-based probabilities to the edges. Neither Wang et al.'s nor Tonella and Ricca's navigation model generation is completely automated. Both require that the values to be input into forms are known beforehand. Another limitation of both approaches is that neither seems to explicitly handle navigation that may depend on different application state (e.g., if a search fails to find any matches because of the contents of the database).

Using user access logs. Kallepalli and Tian [3] and Sant et al. [5] generate *statistical usage-based navigation models* from user access logs. Benefits of using user access logs include that generating the model can be automated and is implementation-independent: every request that the user makes is recorded in the access log, regardless of whether the request was statically or dynamically generated from any programming language source. These models implicitly represent navigation that depends on different application state if users navigate the application that way. Another benefit is that the information from the user access logs can be used to model what users do—for example, how frequently users go from page A to page B versus to page C and how often users start using the application from the home page versus some other entry point. A possible drawback of this approach is that the model is not complete unless users access all parts of the application. However, the goal of this kind of testing is not completeness but to focus on actual usage.

In this paper's study, we focus on usage-based statistical navigation models because we believe the benefits of the usage-based model (representation of usage, automated, implementation-independent model generation) outweigh its limitations (incompleteness).

2.3. Usage-based Statistical Navigation Models

In this paper, the usage-based statistical navigation models (which we will refer to as simply "navigation models" for the remainder of the paper) are based on the Sant et al. approach to

DOI: 10.1002/stvr

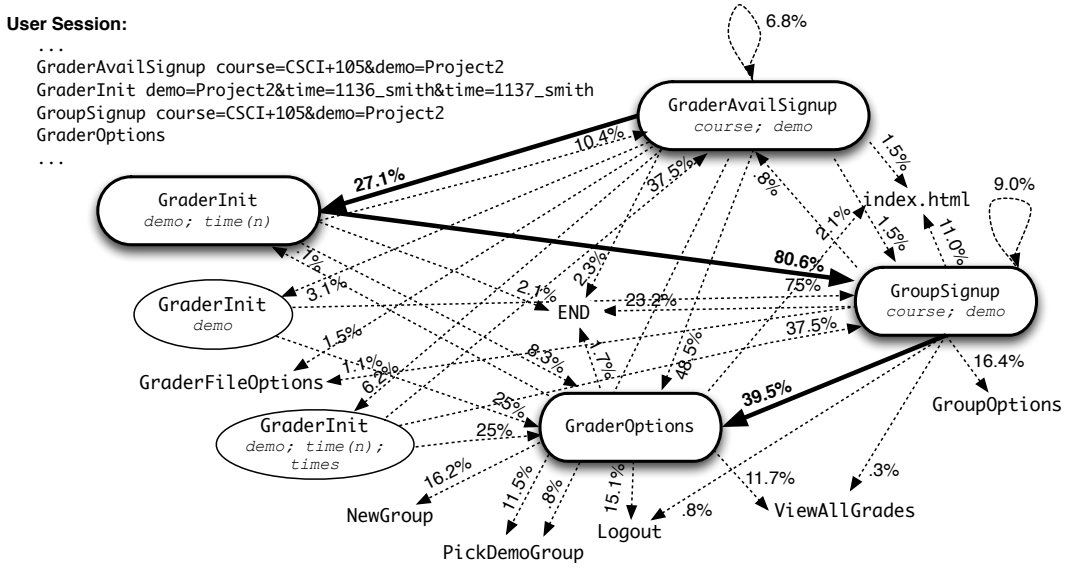


Figure 1. Partial 2-gram Navigation Model for a Set of User Sessions

generating test cases from a statistical model of user sessions [5]. We do not use Kallepalli and Tian's approach because they developed their approach for static web applications and do not fully model dynamic web applications.

The statistical model of user behavior is based on user sessions. User sessions are created by parsing and segmenting user requests to a web application. Each *user session* is a sequence of user requests in the form of base requests and name-value pairs. A request recorder treats hidden parameters the same as regular parameters. 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 30-minute gap between two requests from a user to be equivalent to a session timing out [9].

The *navigation model* (called the "control model" in Sant et al. [5]) is an n-gram Markov model of the user sessions' requests, where n is one more than the number of previous requests used to predict the next request. A navigation model where n=1 (called 1-gram or *unigram*) models the probability of a user making a request, regardless of previous requests. For $n \geq 2$, the navigation model is a directed graph, where the states are the set of n-1-length request sequences in a set of user sessions. A state represents a particular request sequence and its outgoing edges are labeled with the conditional probabilities of the next request. The generated model may have several "start" states, since a user's session may start from any page, e.g., a bookmarked page, a page found by a search engine, or from continuing a timed-out session. In our preliminary studies, users entered an application using as many as 32 different pages—an important phenomenon that is not represented in the non-usage-based models. An "Init" state with directed edges to each of these "start" states is included in the model to represent this usage [5]. Similarly, each of the states representing the last request sequence in a user session has an edge to an "End" state.

Figure 1 depicts part of a 2-gram (or *bigram*) navigation model derived from a set of collected user sessions to a deployed application, highlighting the requests from the partial user session in the upper left. The states represent the set of (length-1) user requests. For simplicity, we do not include the request type in the user session or in the states' labels in Figure 1. The bolder states represent the requests from the partial user session. The edges represent transitions between sequential requests and are labeled with the probability of a user making the second request, given that they made the first request. The thicker edges represent the transitions between requests from the partial user session. For example, given that a user has made a request for `GroupSignup course;demo`,

DOI: 10.1002/stvr

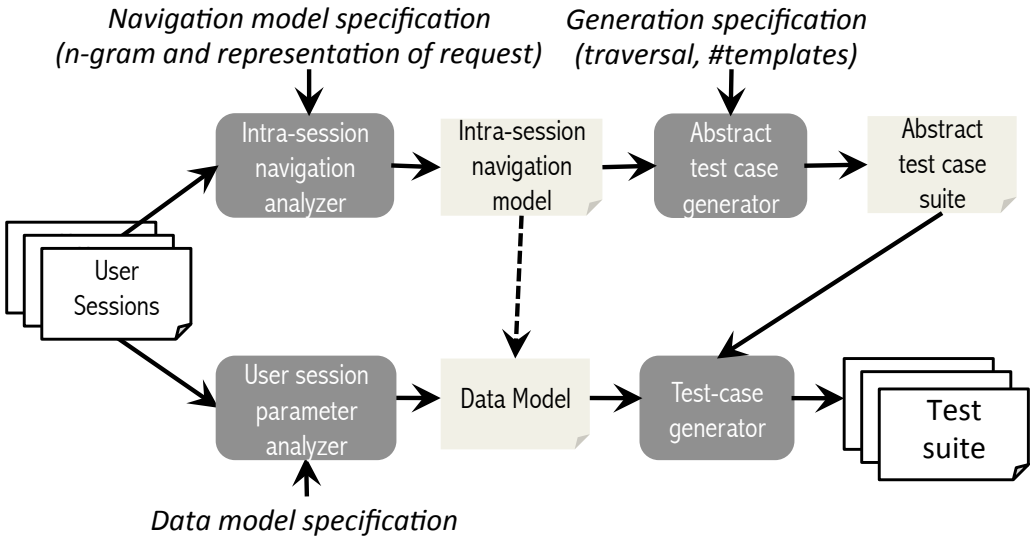


Figure 2. The Statistical Usage-based Test Case Generation Framework

a user has a 39.5% chance of requesting `GraderOptions` and a 16.4% chance of requesting `GroupOptions`.

3. STATISTICAL USAGE-BASED TEST CASE GENERATION PROCESS

Figure 2 shows an overview of the modularized, highly customizable statistical usage-based test-case generation process framework that we have developed. From a set of user sessions and a navigation model specification, the *intra-session navigation analyzer* constructs an intra-session navigation model. We implemented the intra-session navigation analyzer in Python. To represent the navigation models, we utilized `pydot` [10], the Python interface to GraphViz's DOT language [11]. The *abstract test case generator*, also implemented in Python, uses the navigation model and abstract test case criteria to produce a set of abstract test cases. The abstract test cases are input into the *test-case generator* with the data model to output a set of test cases—the executable test suite. The data model is constructed through an analysis of the user sessions. Various data models can be used to generate parameter values [4, 5]. A tester can generate many test cases from one abstract test case by adding parameter values generated from different data models (or even the same model, if it is nondeterministic) to the test cases.

The modularized framework allows for easy configuration of the components using their specifications. Configuring the components for the most effective testing is an open question. In the remainder of this paper, we focus on the question of how to configure the intra-session navigation analyzer for improved testing.

4. OPEN QUESTIONS ON NAVIGATION MODELS IN STATISTICAL MODEL-BASED TESTING

There are several key factors involved in building a usage-based navigation model and generating abstract test cases from the model. We focus on open questions on configuring the navigation models and navigation analyzer because the model forms the basis of the executable test cases. The factors raise open questions, which we discuss in this section, and in Section 5, we describe how we explored these important questions.

DOI: 10.1002/stvr

4.1. Factors in Constructing Navigation Models

Figure 1 represents one way to create the navigation model. We pose questions about alternative configurations and their tradeoffs.

(a) **How should user requests be represented in the model?**

An HTTP request consists of its *request type* (typically `GET` or `POST`), a *resource* (the 'R' in 'URL'), and optional parameter name-value pairs. For example, in the request `GET GraderOptions?user=smith&course=CISC101`, `GET` is the *request type*, `GraderOptions` is the *resource*, `user` and `course` are the *parameter names* and `smith` and `CISC101` are the *parameter values* for the parameter names `user` and `course`, respectively.

Sant et al. [5] represent a request by its **r**equest type, **r**esource, and parameter **n**ames of the data, which we refer to as RRN. Wang et al. [6] use a similar representation but do not explicitly include the request type.

Sant et al. [5] and Wang et al. [6] argue that representing requests by the resource and parameter names balances two concerns: (1) the ability to represent user navigation accurately and (2) maintaining the model's scalability as the number of user sessions and/or the size of the application increases. If we instead chose to represent requests by the resource alone, the size of the navigation model decreases (e.g., there would only be 1 state representing `GraderInit` instead of 3 states in Figure 1), but we would then need to design additional models of the request type and the parameter names to generate executable test cases. Another option is to include parameter *values*, in which case the model may become too large and restricts the generated test cases to the data in the collected user sessions; a tester may not want to be restricted in that way.

(b) **How much history (i.e., what value of n) should be used in the n-gram Markov model of user session requests to predict the next request?**

In an n-gram model, n is one more than the number of previous requests used to predict the next request. Intuitively, one would expect that as n increases, the navigation model would better represent users' navigation. However, as n increases, the size of the model will also increase because the possibilities for $n-1$ sequences increases. There is also a danger of the model overfitting the user session data when n is longer than the length of the longest common subsequence of the user sessions [12] and, thus, the model will not represent any usage beyond that seen in the user sessions. The tradeoffs of n have not been experimentally explored. From Sant et al.'s experiments [5], it is not clear if there is a benefit to using n greater than 1, with respect to code coverage, which is a surprising result. However, their experiments were limited to one small application and only looked at $n \leq 3$, and they did not analyze the effects of the navigation model and the data model separately.

(c) **How many user sessions should be used to build the model and how does the model change as more user sessions are used?**

As user sessions are added to the navigation model, the size of the model may grow if the additional user sessions contain requests to resources or sequences of requests not accessed in the previous requests. We explore how the model in various configurations changes as user sessions are added to the model.

(d) **How does the model topology vary with different configurations for building the model?**

The model's topology—for example, the distribution of the number of out edges from nodes—provides useful information about how users access an application and if usage is indeed useful for testers to model.

DOI: 10.1002/stvr

(e) **How should the transitions between user requests be approximated in the model, i.e., how should the edges in the navigation model be labeled?**

The *abstract test case generator* could generate abstract test cases using the Sant et al. [5] approach: taking a weighted random walk of the navigation model according to the probability distribution learned from the user sessions and adding each visited state's last request to the abstract test case. Tonella and Ricca [1] suggest generating all possible paths through the model and ordering the paths by their probabilities. Since, in general, we found our models were too large and complex (e.g., cycles) to generate all paths, we do not explore Tonella and Ricca's approach empirically. An alternative using the weighted random walk is to consider the edges as all having equal probabilities, instead of using the usage-based probabilities. We explore if using equal probabilities is significantly different from usage-based probabilities.

4.2. Generated Abstract Test Cases

Different configurations for building the usage-based navigation model will most likely affect the characteristics of the abstract test cases generated from the model. Specifically, we note two open questions of concern to testers:

(a) **How representative are the abstract test cases of the original user sessions?**

To evaluate how well the generated abstract test cases represent user sessions, we focus on the abstract test cases' coverage of sequences of requests from the original user sessions, which represents the users' navigation behaviors. A resource often maps to some application code. Regardless of the data model applied, the set of test cases that can be generated is dependent on the abstract test cases. If no abstract test case is generated with a specific sequence of resources (regardless of their representation), then *no test cases with that sequence will be generated* and, thus, will not cover any code associated with that sequence of resources. Therefore, we investigate the sequence coverage, of various lengths, provided by the abstract test cases generated from different navigation models and how probable the generated abstract test cases are with respect to the observed web application usage.

(b) **Do the abstract test cases enable testing of usage not found in the user sessions?**

With these usage-based navigation models, the goal is not to exhaustively generate all possible combinations of requests (as happens for $n = 1$), which is likely to only execute error code, but instead to generate new sequences that are likely to occur based on usage. For example, since a user accessed page A then B in one user session, and a user accessed page B then C in another user session, accessing A then B then C seems to be a viable sequence of requests. We investigate what new test sequences are created using various model configurations.

5. EMPIRICAL STUDY

We designed an empirical study to explore some of the questions surrounding the factors in building usage-based navigation models and the effects on the resultant abstract test cases. Specifically, our empirical study seeks to answer the following questions:

1. **What are the most practical approaches to constructing usage-based navigation models for test case generation?** (Questions a-e in Section 4.1.)
2. **What are the characteristics of the generated abstract test cases for different model configurations and how can they be tuned for different testing goals?** (Questions a and b in Section 4.2.)

5.1. Subjects

In this paper, we target web applications written in Java using servlets and JSPs. The applications consist of a backend data store, a Web server, and a client browser. Since our user-session-based

DOI: 10.1002/stvr

Subject	# of Classes	# of NCLOC
Masplas	9	609
Book	11	5279
CPM	76	7430
Logic	106	10704
Logicv2	135	16491
DSpace	291	29430

Table I. Subject Application Characteristics

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 13 subject user-session sets from user requests to 5 publicly deployed applications. The applications were of varying sizes, technologies, and representative of web application activities and usages: a conference submission and registration website (Masplas); an e-commerce bookstore (Book) [13]; a course project manager (CPM); an online symbolic logic tutorial (Logic and a significantly revised version, Logicv2); and a customized digital library (DSpace) [14]. Book is the same application used in Sant et al.'s evaluation [5]. Table I summarizes the applications' code characteristics; specifically, column 1 presents the number of classes and column 2 presents the number of non-comment lines of code (NCLOC).

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. [5]. We collected accesses for each application over a long period of time: Masplas: 2 months, CPM: 5 academic semesters, Logic: 2 academic semesters, DSpace: 3.5 years.

We converted the user accesses into user sessions using Sprenkle et al.'s framework [9]. Before processing user accesses, we removed accesses from IP addresses that are known to be spiders, bots, or malicious to reduce the noise from non-users and better create models of human users' navigations. We also remove requests for the embedded static content, such as images, cascading style sheets (CSS), and JavaScript (JS) files, that are requested when a page loads. For CPM and DSpace, we partitioned the user sessions by the time periods in which they were collected to provide more sets of user session subjects to model and compare.

Table II shows the characteristics of the collected user sessions, in terms of the number of user sessions (totalling over 19,000 sessions), the number of user requests (totalling nearly 129K), and the percent of application code covered by the user sessions using Cobertura [15]. We report line coverage to show that the user sessions cover a large portion—but not all—of the application. There are several reasons for what may seem like lower coverage than might be expected. The primary reason stems from the way coverage was collected: the JSPs were compiled into Java classes that were instrumented to collect coverage. Common JSPs that were included in multiple JSPs (e.g., navigation/sidebar code) were statically included in each compiled Java class. The common JSPs were not completely covered in each of the JSPs in which they were included but were generally covered across all included JSPs. The second reason is specific to DSpace: DSpace has several different configurations, which correspond to the code executed. We chose to include rather than remove the code for the alternative configurations to not artificially inflate the code coverage.

In Table II, we see differences among the collected user sessions for the same application in terms of code coverage. While CPM's usage was generally the same for each semester—creating new user accounts, students and graders interacting, DSpace's usage varied over time. When DSpace was first deployed, the application administrators had to spend more time configuring it and making changes through the web interface. The number of administrative tasks decreased as the application's desired configuration became more stable.

5.2. *Navigation Model Analysis*

In this section, we investigate the open questions in constructing navigation models of web applications (Section 4.1, questions (a) to (e)). We discuss the results from studying how to represent

DOI: 10.1002/stvr

Subject	# User Sessions	# Requests	% Lines Cvd
Masplas	169	1107	89%
Book	125	3564	61%
CPM1	58	1326	47%
CPM2	203	2393	66%
CPM3	105	1528	50%
CPM4	168	2240	58%
CPM5	356	4865	54%
Logic	497	16,179	80%
Logicv2	374	16,052	78%
DSpace1	1087	12,277	74%
DSpace2	5012	14,110	46%
DSpace3	3853	15,126	45%
DSpace4	7687	38,155	49%
Total	19,694	128,934	–

Table II. Characteristics of User Session Sets

user requests and inter-request navigation and then factors affecting model size. We generate the variations of navigation models for investigation by running the *intra-session navigation analyzer* (Section 3) on each set of user sessions.

5.2.1. Methodology. To answer question 4.1(a), we generated navigation models using 3 different representations of user requests: requestType+resource (RR), requestType+resource+parameter names (RRN), and requestType+resource+parameter names+parameter values (RRNV). We investigated the model characteristics for n from 1 to 10 to answer question 4.1(b). Across all our sets of user sessions, only Masplas had a maximum longest common subsequence less than 10. In the worst case (DSpace, $n = 10$), the scripts generated each model in at most 5 minutes.

To answer question 4.1(c), we generated navigation models incrementally to analyze how much the navigation model's size changes as user sessions are added, to help determine when a tester could stop collecting user sessions. We performed three experiments, adding user sessions (1) in collection order, (2) in order of decreasing number of requests, which is an estimate of the best case because larger user sessions are more likely to contain requests that have not been seen before, and (3) in order of increasing number of requests, which is an estimate of the worst case. Note that the order does **not** affect the final resulting model.

For each n, we measured the size of the navigation models in terms of their number of states and edges, counted the number of out edges from the states in the models, and counted the number of times the states' represented sequences were accessed in the original user sessions to answer question 4.1(d) and computed statistics about the probabilities labeling the edges to answer question 4.1(e).

5.2.2. Results. In this section, we present the results of our empirical study, which help us identify the most practical approaches to constructing usage-based navigation models for test-case generation.

(Q. 4.1.a) Representing User Requests. Table III shows the number of states and edges in the models generated using n=2, when representing user requests using the requestType+resource (RR), requestType+resource+*parameter names* (RRN), and requestType+resource+parameter names+*values* (RRNV) and the percent increase in both states and edges when adding information (parameter names and values, respectively) to the representation. Model growth percentages are to 3 significant figures. The median, mean, and standard deviation of model growth across all 13 generated models are in the bottom row.

The percentage growth when adding parameter names to the representation (the columns labeled **Growth**) is less than 100% for most sets of user sessions, with an average of 159% for states and

DOI: 10.1002/stvr

	Repre-	2-RR		2-RRN		Growth		2-RRNV		Growth over 2-RRN	
	sentation	States	Edges	States	Edges	States	Edges	States	Edges	States	Edges
Set of User Sessions	Masplas	23	218	36	264	56.5%	21.1%	191	489	431%	85.2%
	Book	10	71	27	173	170%	144%	910	1778	3270%	928%
	CPM1	49	185	53	200	8.16%	8.11%	471	860	789%	330%
	CPM2	61	254	98	333	60.7%	31.1%	609	1155	521%	247%
	CPM3	52	184	85	245	63.5%	33.2%	376	723	342%	195%
	CPM4	51	200	80	263	56.9%	31.5%	502	1018	528%	287%
	CPM5	53	220	72	265	35.9%	20.5%	893	1777	1140%	571%
	Logic	74	481	88	547	18.9%	13.7%	3729	7257	4138%	1227%
	Logicv2	83	325	97	387	16.9%	19.1%	3953	7521	3980%	1840%
	DSpace1	65	400	677	1775	942%	344%	4611	7212	581%	306%
	DSpace2	64	309	250	734	291%	138%	2407	4244	863%	478%
	DSpace3	78	365	230	934	195%	156%	3651	5778	1490%	519%
	DSpace4	58	253	147	515	153%	104%	4291	10616	2820%	1960%
					Median	60.7%	31.5%		**Median**	863%	478%
					Mean	159%	81.7%		**Mean**	1610%	691%
					Std Dev	250%	95.9%		**Std Dev**	1420%	621%

Table III. Comparing Request Representation: RR vs. RRN vs. RRNV, for n=2

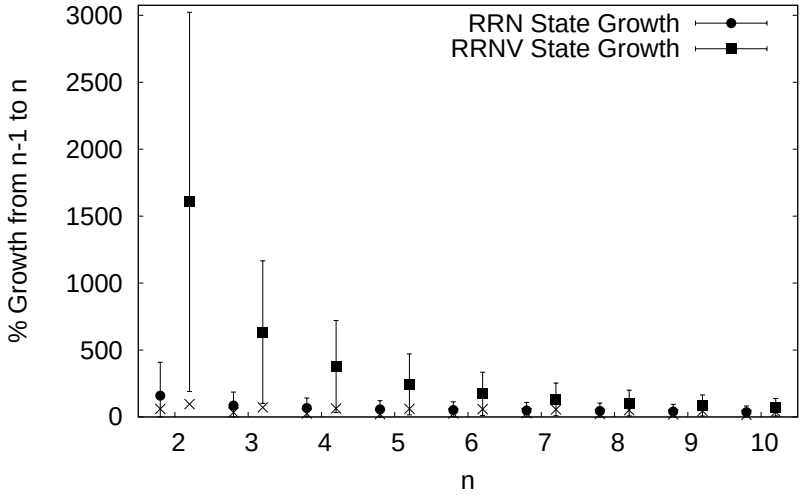


Figure 3. Comparing Request Representation: State Growth in Models from $n-1$ to n for $2 \leq n \leq 10$

81.7% for edges. The growth is larger for the DSpace user session sets because DSpace encodes some parameter names with values (e.g., `item_1234`), which results in many parameter names, e.g., as compared to just using `item`.

The percentage growth when including parameter values (the columns labeled **Growth over 2-RRN**) is much larger, all above 300% for states and 200% for edges, with means of 1600% and 691% for states and edges, respectively. In general, as the number of user sessions increases, the growth percentage will also increase, as there will typically be a greater number of user-supplied unique values for some requests (e.g., a search field).

It is interesting to note that across the 5 sets of user sessions for the CPM application, there are differences in the size of the models, but the order of magnitude of growth is the same across all sets of user sessions. The growth trends are quite different for DSpace. DSpace1 by far has the most growth from RR to RRN, whereas DSpace4 has the most growth from RRN to RRNV. We believe DSpace1's growth in RR to RRN can be attributed to the parameter encoding discussed earlier because the administrative functionality that is prevalent in DSpace1 but not in the other sets of user sessions often uses encoded parameters. DSpace4's requests are dominated by search result pages, which have parameters that allow many different combinations of values. To summarize, the application's usage—rather than the application code itself—had an impact on the growth behavior.

DOI: 10.1002/stvr

The previously described growth results focused on models generated using $n = 2$. Figure 3 shows the growth rate of states from $n - 1$ to n when $2 \leq n \leq 10$. The x-axis represents the n value used to generate a model, and the y-axis represents the percentage growth of nodes in the generated models. We plotted the average node growth for the models generated from all sets of user sessions as points and the standard deviation as error bars, for both the RRN and the RRNV-based models. The median values, represented as an '$\times$', are also plotted for each model.

We observe that as the value of n increases, the average percentage growth for both representations greatly decreases and the differences in growth between adding parameter names (RRN) and parameter names and values (RRNV) decreases. The standard deviation is large for $n = 2$ and decreases as n increases, which indicates that growth rates become more similar as n increases. When the standard deviation is large, the median is much less than the average.

Our results provide empirical evidence for researchers' beliefs that representing user accesses by the resource and parameter names (RRN) provides good representation with reasonable growth [5, 6]. Using RRN, there is typically less than a 100% increase in model size over RR (the median in Table III) and no need to develop a model of the parameter names, with its possible inaccuracies, although there is still a need for a parameter value model—whose time and space costs could vary greatly—to make the test cases executable. However, the percentage increase in model size when including parameter values (RRNV) decreases as n increases; in our results, on average when $n \geq 9$, the increase is negligible with less than 100% increase in model size. For the rest of this paper, we focus on models using RRN as the request representation.

Implication for testing: *When generating test cases automatically through a statistical model-based approach, separate the model of user behavior into two models and the test case generation process into two steps: (1) build a usage-based navigation model that represents the user accesses by resource and parameter name, which is then used to generate abstract test cases and (2) develop a separate data model for adding parameter values and use the data model to generate executable test cases. However, if the tester wants to use a sufficiently large n (application-dependent), using RRNV may not require too many additional resources than using RRN in terms of the model's size and does not require a data model.*

(Q. 4.1.b) Factor Affecting Model Size: Amount of History (n). Figures 4(a) and 4(b) show how the RRN navigation model's size grows as n increases in terms of the number of states and edges, respectively, for each set of user sessions. The x-axis represents the n used to generate the model, and the y-axis represents the number of states or edges in the resulting navigation model, respectively.

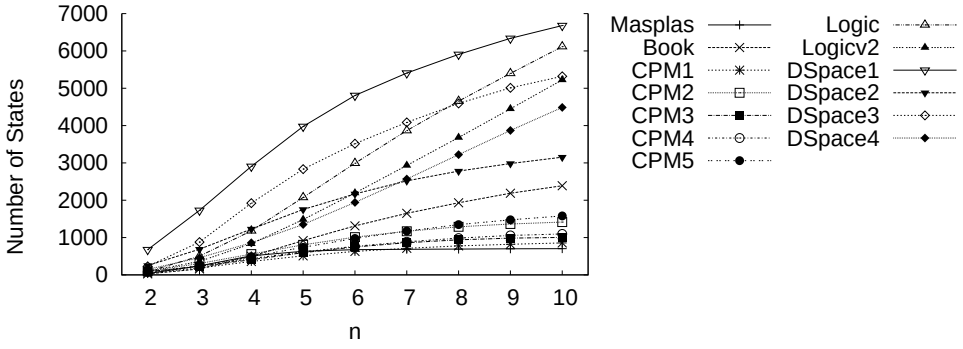
As expected, as n increases, the number of states and edges increases similarly. For most of the models generated, the increase in model size slows down as n increases. The exceptions are Logic, Logicv2, and DSpace4, which contain the largest number of requests and, thus, have the most potential for growth in terms of the number of different sequences and parameter name combinations. Although the exceptions' growth rates may look large, for $n = 10$, the growth rate is only between 12.8% to 17.4%.

While the navigation model size continues to grow, we would like to know that the growth slows down. Figure 5 shows the *change* in the model growth rate between $n - 1$ to n as n increases. The x-axis represents n. The y-axis represents the change in growth rate from $n - 1$ to n—thus, our x-axis starts at 3 instead of 2. We plotted the average state and edge growth for the models generated from all sets of user sessions as points and the standard deviation as error bars for both the RRN-based models. The median values, represented as an '$\times$', are also plotted for each model.

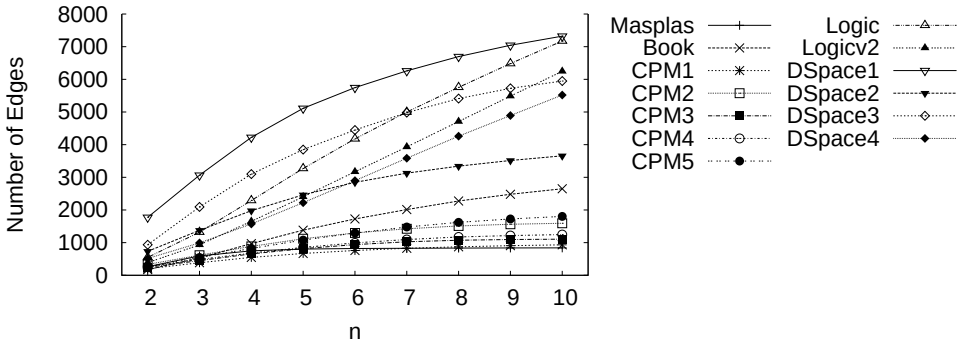
The graph clearly shows that the growth rate decreases as n increases and that there is a significant decrease in growth rate for $n \geq 4$. There is also a large variation in growth rates for the models from $n = 2$ to 3, but the variability decreases as n increases. Edge growth is always smaller than state growth, which is not too surprising because the edges outnumber the states considerably for $n = 2$.

We expect that, as n increases beyond 10, the growth rate will eventually drop to 0 because the lengths of the user sessions and number of RRNs in an application are finite. These results suggest that the navigation models do not grow exponentially in size as n increases.

DOI: 10.1002/stvr



(a) Model Growth In Terms of Number of States



(b) Model Growth in Terms of Number of Edges

Figure 4. RRN Navigation Model Growth as n increases

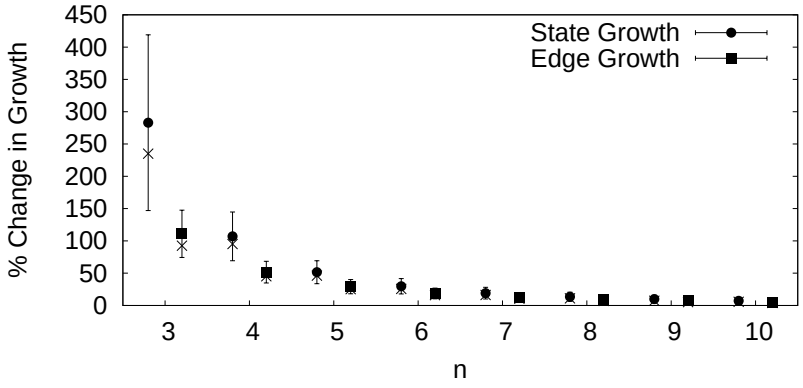
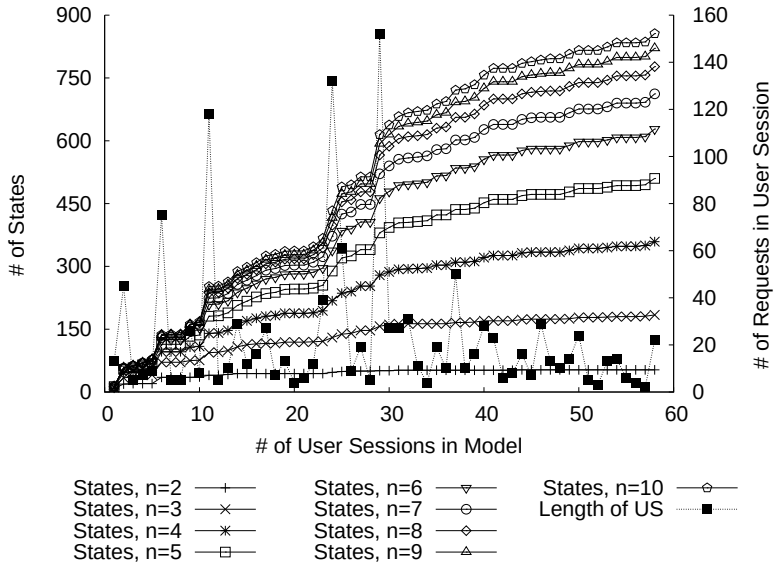


Figure 5. RRN Navigation Model Change in Growth Rate between $n-1$ to n as n increases

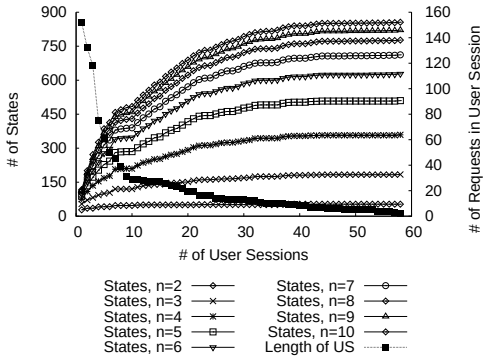
Implication for testing: *Model growth alone is not a key factor for determining which n to use for building the navigation model because increasing n does not result in exponential growth.*

(Q. 4.1.c) Factor Affecting Model Size: the number of user sessions. As user sessions are added to the navigation model, the size of the model may grow if the new user sessions contain requests to resources or sequences of requests not accessed in the previous requests. Figure 6 is representative of all the sets of user sessions as additional user sessions are added to the navigation model in three different orders of processing user sessions. The x-axis represents the user session's id in the respective order: Figure 6(a), in the order the user sessions were collected; Figure 6(b),

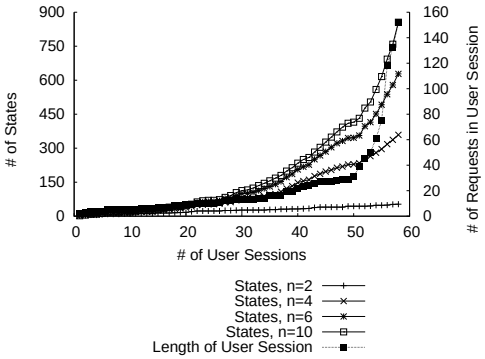
DOI: 10.1002/stvr



(a) User Session Order: Collection Order



(b) User Session Order: Decreasing Test Case Length



(c) User Session Order: Increasing Test Case Length

Figure 6. Navigation Model Growth for CPM1 Using Different Orderings of User Sessions

in approximately the best case order—largest test case length to smallest; and Figure 6(c), in approximately the worst case order—smallest test case length to largest. (We only show a subset of the model results in Figure 6(c) to increase readability of the graph.) The left y-axis represents the number of states in the model, which represents all the user sessions up to and including this user session. The right y-axis represents the length of the user session in terms of its number of requests.

As we hypothesized, adding a longer test case often coincides with increasing the size of the model, e.g., test case 29 in Figure 6(a) shows a distinct increase in model size for $n \geq 2$. Furthermore, the slope of the graph when the user sessions are in decreasing length (Figure 6(b)) order reaches a point where it decreases. We observed this trend across all n and for all sets of user sessions. The result is not surprising because user sessions are known to be redundant [7, 16].

We also observe that the model for $n = 2$ stabilizes faster than models for larger values of n in all three orderings. In Figure 6(a), the 2-gram model seems to stabilize around 25 user sessions, whereas the 3-gram model stabilizes around 35 user sessions. Again, that result is expected because additional user sessions are more likely to have length-$n-1$ sequences that were not seen in previous user sessions than they are to have new length-$n-2$ sequences. In other words, there are more possibilities for unique sequences of length $n-1$ than there are for length $n-2$. For

DOI: 10.1002/stvr

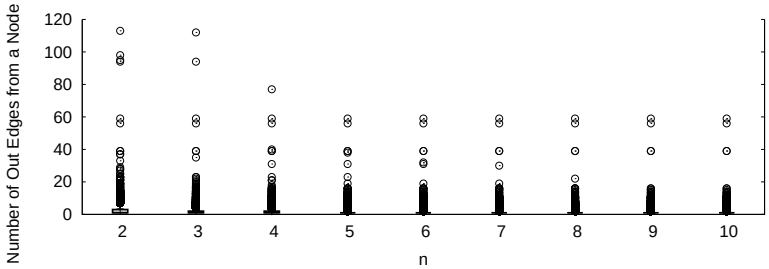


Figure 7. Distribution of Number of Out Edges from States Across All Applications

example, for $n = 2$, a new state means that a new RRN request was made (e.g., `GET newpage`), whereas for $n = 3$, a new state means that a new sequence of RRN requests was made (e.g., `GET page1` followed by `GET page2`).

Implication for testing: *A tester could collect user sessions—ordering them from longest to shortest to observe the stability trends more easily—and stop adding user sessions at any time after the model growth rate declines to a desired threshold and thus save time and space in collecting user sessions because user sessions tend to be redundant. Models generated using larger values of n will likely require larger numbers of user sessions. Resource-permitting, a tester may want to use as many user sessions as possible because additional user sessions provide more accurate estimates of users' transition probabilities, and the tester can throw out the user sessions and keep the smaller model.*

(Q. 4.1.d) Topology of the Navigation Graph. Figure 7 represents the distribution of the number of out edges per state—representing the different options users take from a given state in a model—for all states in all models generated from the user session sets. The x-axis represents the value of n used to generate the model, and the y-axis represents the number of out edges.

As n increases, the number of out edges decreases. This result is not surprising because, given that a user has done a k-length sequence of events (e.g., completing a multi-page task), it is less likely that the user will have many options for the $k + 1$ step. Furthermore, it's also less likely that other users have performed the same k-length sequence of events—leaving the potential number of $k + 1$ steps smaller, i.e., if only one person performs the $k + 1$-length sequence, there will be only one out edge from the state representing the k-length sequence. These results coincide with the reduced model growth rate as n increases.

Many of the outliers for $n \geq 5$ are caused by the Init state, which represents the beginning of all user sessions for an application. User sessions may start at many different pages, for example, the application's home page, a page found during a search, a bookmarked page, or an internal page being returned to after a period of inactivity, which constitutes a new user session. Figure 7 highlights the diversity in users' initial application access and points to a limitation of models that assume one start page and, thus, don't represent the variety of ways a user may use the web application.

We were surprised that the median number of out edges from states within all n-gram models was only one. As n increases ($n \geq 5$), the 75^{th} percentile is also 1. In Table IV, we examine the states with only one out edge across all models. The first column is n and the second is the total number of states in all the models generated for that n. The third and fourth columns show the number and percentage of the total number of states that have only one out edge, respectively. For even $n = 2$, the majority of states have only one out edge, and the percentage increases up to 91% for $n = 10$. These results seem to imply that, in general, users do not follow many different paths out of a state.

The states with only one out edge tend to represent sequences in a few broad, non-exclusive categories: parts of forms that have obvious next steps, e.g., confirming what the user did in the previous step, and rarely accessed sequences—some of which were erroneous, unintended sequences by the developers—which therefore have few possibilities for out edges.

For $n = 2$, the states with only one out edge tend to be pages that set up multi-page forms, and users simply follow to the next step, e.g., confirming what they did in the previous step.

DOI: 10.1002/stvr

n	# of States	States w/ Only 1 Out Edge #	% of Total
2	1953	1039	53.2%
3	6224	3892	62.5%
4	12205	8851	72.5%
5	18295	14489	79.2%
6	23746	19838	83.5%
7	28530	24604	86.2%
8	32808	28923	88.2%
9	36650	32926	89.8%
10	40030	36448	91.1%

Table IV. Statistics about States with Only One Out Edge

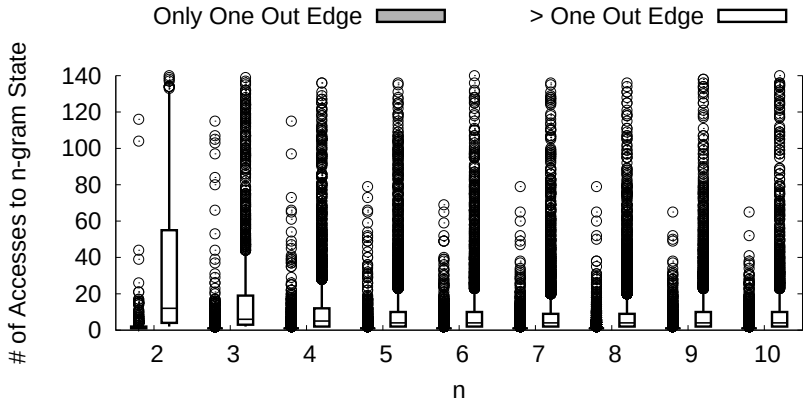


Figure 8. Comparing States with One Out Edge and States with More Than One Out Edge: Distribution of Number of Accesses to n-gram States Across All User Sessions. We restricted the y-axis to 140 accesses; there were states with more than one out edge that were accessed more than 140 times in the user sessions.

DSpace has some one-out-edge states with different characteristics than the other applications. Many of the one-out-edge DSpace states have data hardcoded into the parameter names, e.g., `bitstream_description_353_384`, which means that the request is specific to one particular item in the application's data base. Also in DSpace, the order of the values for certain parameters matters. For example, the authors of a publication are passed in using parameter names identified by a number, e.g., `contributor_author_last_4`. In both of these cases, the states identified by the RRN sequence become specific to particular items and, thus, there are not many accesses to the item to create the possibility for many out edges.

To better understand users' aggregate navigation behavior with respect to states with only one out edge and states with more than one out edge, we plotted the distribution of the number of times the user sessions access the RRN sequence represented by a state in Figure 8. Recall that a state represents a sequence of RRN requests. To count the number of occurrences of a 3-gram state in the user sessions, we count the number of times the length-2 RRN sequence occurs in the set of user sessions.

The x-axis represents the value of n used to generate the model. The y-axis represents the number of times a state is accessed in the user sessions. The box represents the middle 50% of the data (the inner quartile range IQR), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The center horizontal line within each box denotes the median. We limited the y-axis maximum to 140 to increase readability of the graph and more clearly show the differences between the states with only one out edge and those with multiple out edges. The distribution for states with only one out edge are in gray and on the left while the distribution for states with more than one out edge are on the right.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

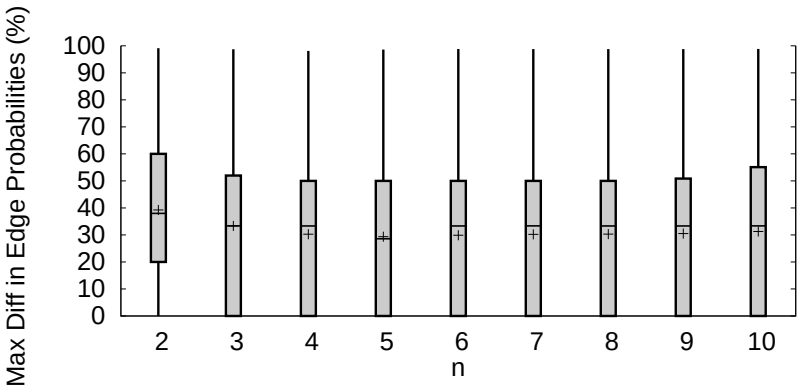


Figure 9. Distribution of Maximum Difference in Edge Probabilities Across All Applications

Figure 8 indicates that states with more than one outgoing edge occur much more frequently than states with only one outgoing edge. The median number of occurrences for states with only one out edge was 1, for all n, and the averages were less than 2.1, while the median for states with multiple out edges was between 12 (for $n = 2$) and 4 ($n \geq 5$), while the means ranged from 22.5 ($n = 10$) to 135 ($n = 2$). The means are difficult to see in the figure because they are mixed in with other data points.

It is interesting to note that the most frequently occurring 3-gram state is two consecutive requests to view items from a search (`results.jsp`), which happens 11,914 times in DSpace4 and 15,283 times overall. There is no link on the `results.jsp` page to itself; instead, a user must use the back button to navigate back to the search results page and then select another publication to view. Therefore, this sequence would not be represented in a statically determined navigation model.

Combining the results from Table IV and Figure 8, it seems that the most common user behavior occurs where there is more than one out edge. This observation matches expectations: commonly accessed sequences are more likely to be followed by a diverse number of requests than less commonly accessed sequences.

Implication for testing: *Usage information is important to model: users tend to access the web application in ways that are not readily captured by statically determined navigation models. For example, usage exposed behavior that static models do not represent: users start accessing web applications from a variety of resources, which cannot be determined statically, and users sometimes access resources not through links or forms.*

(Q. 4.1.e) Representing Inter-Request Navigation. To answer the question about how transitions between user requests should be approximated in the model, we analyzed the range in the probabilities computed from usage information for each node's out edges, and compared with equal probabilities on its outgoing edges. For all nodes with at least two outgoing edges, we identified the edges with the minimum and maximum probabilities, respectively, and took the difference between those probabilities. The results of the edge analysis for all applications are in Figure 9. The x-axis represents the value of n used to generate the navigation model. The shaded box represents the middle 50% of the data (the inner quartile range IQR), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The horizontal line denotes the median and $+$ represents the mean.

For half of the nodes (the median), the most likely edge is favored at least 25% more than the least likely edge (between 25.0% and 35.4%). Furthermore, for 25% of the nodes (the upper horizontal line in the shaded box), the most likely edge is favored at least 46% more than the least likely edge. The averages ranged between 27.1% and 38.6%. The maximum difference is larger for n=2. We did not observe any other distinct trends as n increases.

To confirm our intuition that usage is not equally distributed, we performed a statistical analysis of the usage frequency distribution computed from usage information for each node's out edges

DOI: 10.1002/stvr

Application	n 2	3	4	5	6	7	8	9	10
Book	100.0%	100.0%	97.1%	96.7%	85.2%	78.9%	85.7%	75.0%	88.9%
CPM1	89.5%	91.3%	94.4%	100.0%	92.3%	90.9%	88.9%	88.9%	90.0%
CPM2	100.0%	84.4%	88.9%	86.4%	77.8%	92.9%	92.3%	92.9%	92.9%
CPM3	95.0%	86.4%	75.0%	81.8%	80.0%	80.0%	80.0%	80.0%	80.0%
CPM4	100.0%	100.0%	91.7%	86.4%	94.7%	88.9%	89.5%	94.1%	94.1%
CPM5	90.9%	87.9%	82.9%	84.1%	89.5%	86.1%	90.9%	93.1%	96.3%
DSpace1	100.0%	96.7%	100.0%	90.8%	90.5%	89.1%	88.7%	77.2%	77.2%
DSpace2	90.5%	87.9%	84.6%	87.9%	90.6%	86.0%	82.7%	84.0%	83.3%
DSpace3	92.1%	93.3%	86.1%	86.1%	72.2%	77.9%	73.5%	74.6%	77.2%
DSpace4	90.9%	95.6%	90.7%	90.9%	92.1%	92.3%	84.8%	84.8%	80.0%
Logic	97.3%	85.7%	87.1%	78.5%	82.2%	82.8%	86.6%	85.6%	80.0%
Logicv2	100.0%	92.5%	84.0%	83.8%	81.0%	81.3%	87.3%	85.5%	80.9%
Masplas	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Total	95.4%	91.9%	88.7%	86.7%	85.4%	85.7%	85.3%	84.1%	81.5%

Table V. Percentage of Nodes with More than 1 Out Edge with a Sufficient Number of Observations Whose Edge Distributions Are Not Equivalent to Uniform Distributions Using χ^2 Goodness-of-Fit (Rejected Null Hypothesis with $p \leq .05$)

Application	n 2	3	4	5	6	7	8	9	10
Book	83.3%	34.7%	17.2%	12.8%	11.6%	9.1%	6.6%	6.4%	5.3%
CPM1	43.2%	26.7%	17.8%	14.0%	15.3%	14.3%	14.5%	16.1%	21.3%
CPM2	43.1%	27.1%	17.8%	12.9%	10.7%	9.3%	9.8%	11.8%	13.2%
CPM3	55.6%	25.9%	13.4%	8.8%	9.0%	10.3%	12.3%	15.2%	16.9%
CPM4	46.7%	28.3%	21.4%	18.3%	16.2%	15.7%	18.1%	19.8%	21.5%
CPM5	44.0%	28.0%	25.5%	22.7%	19.7%	18.8%	18.5%	18.0%	19.3%
DSpace1	19.7%	16.2%	12.8%	13.0%	14.3%	15.6%	15.3%	15.4%	16.8%
DSpace2	48.3%	25.2%	20.2%	16.8%	15.3%	18.2%	18.2%	18.2%	18.2%
DSpace3	37.3%	16.1%	13.8%	15.2%	16.2%	13.3%	13.8%	15.1%	16.2%
DSpace4	40.2%	38.2%	35.0%	31.0%	31.6%	30.9%	32.2%	32.6%	31.8%
Logic	49.3%	26.3%	18.8%	18.4%	16.3%	17.5%	17.2%	17.6%	18.2%
Logicv2	36.9%	29.3%	25.1%	23.7%	23.1%	21.1%	20.2%	20.1%	19.8%
Masplas	18.2%	3.0%	4.0%	4.9%	6.2%	4.8%	4.8%	4.9%	5.0%
Total	38.2%	23.3%	18.7%	17.9%	18.0%	18.0%	18.3%	19.1%	19.8%

Table VI. Percentage of Nodes with More Than 1 Out Edge That Have a Sufficient Number of Observations to Perform a χ^2 Goodness-of-Fit Test

and compared that to a uniform distribution. For all nodes with at least two outgoing edges, we performed a χ^2 goodness-of-fit test on the frequencies of the outgoing edges. In Table V, we report the percentage of nodes where we rejected the null hypothesis that the usage distribution was uniform using $p \leq .05$ for each application and across all applications. The percentages for the nodes with edge distributions considered non-uniform does not include nodes that did not have a sufficient number of observations to allow a χ^2 goodness-of-fit test, i.e., the expected frequency for some edges was less than 5 uses. Figure 8 already showed that a large number of nodes were accessed infrequently. The percentage of nodes that had a sufficient number of observations to perform the χ^2 goodness-of-fit test is in Table VI. The nodes in Table VI represent the most commonly accessed nodes/functionality and, thus, are the nodes for which accurate representation of usage matters most.

Table V shows that we rejected the hypothesis of uniformity far more often than we accepted it. The rejection percentage varies from a minimum of 72.2% (DSpace3, $n = 6$) to a maximum of 100% for 17 combinations of user session sets and values of n. One trend is that, as n increases, the percentage of nodes where we rejected the null hypothesis for each application model usually decreases. The likely reason for the decrease is that there are fewer observations of longer usage

DOI: 10.1002/stvr

sequences, which means that we will have less confidence to say that the distribution is likely not uniform.

These results suggest that actual web application usage is considerably different from estimating user behavior without gathering usage information, such as by using equal probabilities on transitions. There is a caveat, however: the results in this section only include states that have more than one out edge. As n increases, the percentage of all the states represented in this figure decreases. It appears that usage probabilities may matter less as n increases because there are more states that have only one out edge to another state.

Implication for testing: *While generating abstract test cases using a navigation model built without usage-based transition labels avoids the potential challenges of user log capture and analysis (e.g., privacy concerns or data collection effort), the resulting navigation model and abstract test cases will not align well with actual user behavior. Thus, if the tester's goal is to execute tests closely representing user behavior or prioritizing tests based on user behavior, the tester should gather usage logs and build a navigation model using those logs.*

5.3. Abstract Test Case Analysis

This section presents our study of the generated abstract test cases from different variations of navigation models. We focus on the questions posed in Section 4.2: how representative the abstract tests cases are of the user sessions and how much they cover new navigations not in the original user sessions.

5.3.1. Methodology. We used the *abstract test case generator* (Section 3) to generate abstract test cases for evaluation. To account for the randomness of the random walk through the navigation model, the abstract test case generator generated 30 suites containing 500 non-duplicate abstract test cases for each navigation model. If the random weighted walk generated a duplicate abstract test case (i.e., the same sequence of RRN requests as a previously generated abstract test case), the abstract test case was discarded. For the 1-gram model, we set the number of requests in a test case to 15 requests, based on the average sizes of the original user sessions. For $n > 1$, the number of requests in an abstract test case was determined by the weighted random walk of the navigation model. We generated a fixed number of abstract test cases instead of using a URL-coverage-based stopping criteria [17] so that we could compare results across applications and aggregate results without introducing bias towards a user session set. In the worst case, our abstract test case generator took 8 minutes to generate 500 non-duplicate abstract test cases.

To answer question 4.2(a) about how representative the abstract tests cases are of the original user sessions, for each abstract test case suite, we measured the number of the user sessions' RRN sequences of various lengths that are represented by the abstract test case suite. We also generated suites that contained 100, 200, 400, and 800 abstract test cases from the n-gram models (where $2 \leq n \leq 10$) and calculated the sum total probability of the generated test case suite. In general, larger probabilities indicate that the test suite represents the user sessions' common behavior well. To calculate the probability of a test case, we multiply the probabilities of each traversed model edge used to generate the abstract test case together, as in Tonella and Ricca's paper [1]. We generated 30 test suites to account for the randomness in generating a suite.

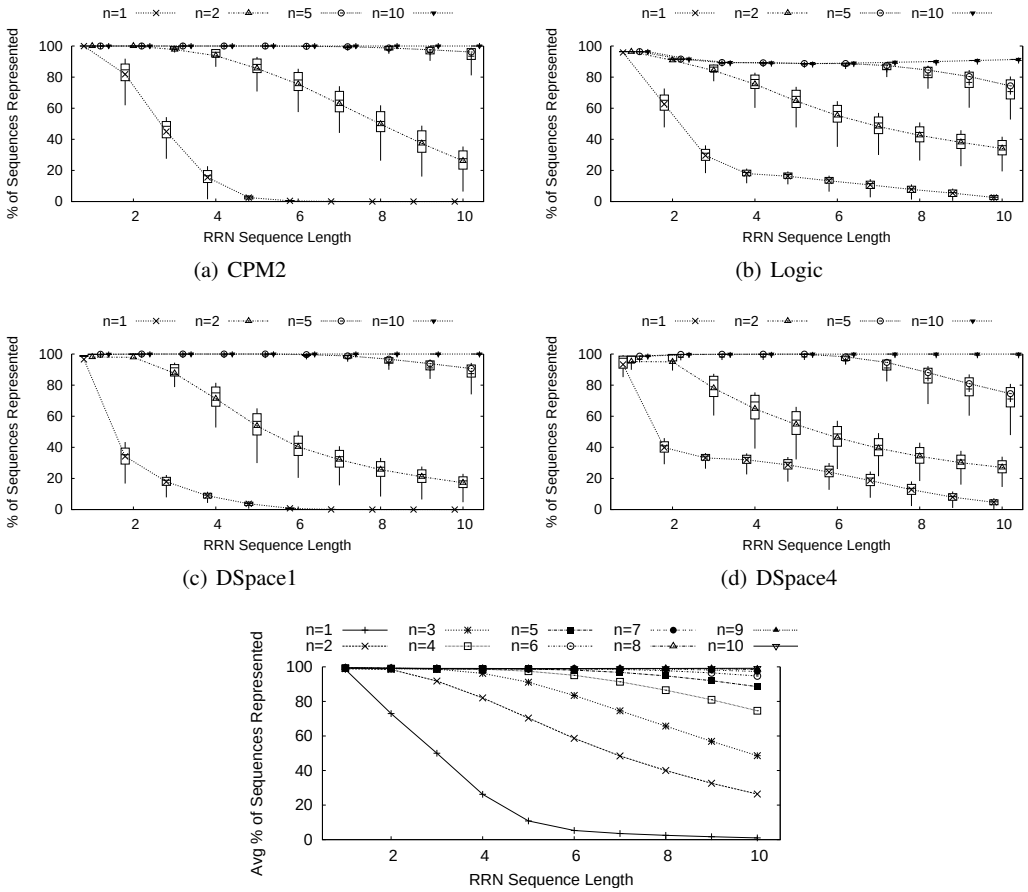
To answer question 4.2(b) about what new usage—beyond the user sessions—the abstract test cases enable, we measured the number of new, unique RRN sequences of various lengths that the abstract test cases explore.

To help put the previous results into context, we measured the number of requests in each generated abstract test case.

5.3.2. Results. In this section, we present the results of our empirical study, which help us to tune the generated abstract test cases for various testing goals.

(Q. 4.2.a) Representation of User Sessions: RRN Sequence Coverage. Figure 10 shows the percentage of the user sessions' RRN sequences of various lengths that are represented in the abstract test case suites generated using various n-gram models. The x-axis represents the length

DOI: 10.1002/stvr



(a) CPM2 (b) Logic

(c) DSpace1 (d) DSpace4

(e) Average Across All Abstract Test Case Suites

Figure 10. Percent of RRN Sequences Represented in Abstract Test Cases

of the RRN sequence. The y-axis represents the percentage of the length-x RRN sequences in the user sessions that are represented in the abstract test cases. The lines show the data for the test cases generated from the various n-gram models. Figures 10(a) through 10(d) show representative results for the abstract test cases generated from models of four sets of user sessions for n =1, 2, 5, and 10. The box represents the middle 50% of the data (the inner quartile range IQR), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The horizontal line denotes the median and + represents the mean. To improve readability of the graph, we do not show outliers. Figure 10(e) shows the average across all applications for $1 \leq n \leq 10$.

In general, as the sequence length increases, the percent coverage decreases, for all sets of abstract test suites. This result is not surprising since a test suite can never cover more length-k sequences than length-k-1 sequences. Furthermore, as the n used to generate the model increases, the percentage of sequences covered by tests increases. We observe the same trend for all sets of user sessions. As n increases, the more history is used to predict the next request; therefore, test cases generated from an n-gram model must be made up of sequences of length n-1 from the user sessions.

For all sets of abstract test suites for n=10 except for those for Logic and DSpace, the test suite covers 100% of RRN sequences of lengths 1 to 10. 100% coverage was surprising because we generated the abstract test cases using a weighted random walk and did not do anything to guarantee coverage. Logic and DSpace have larger (Figure 4), more complex navigation models, which we believe is one reason that the test cases for n=10 do not always have 100% coverage.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

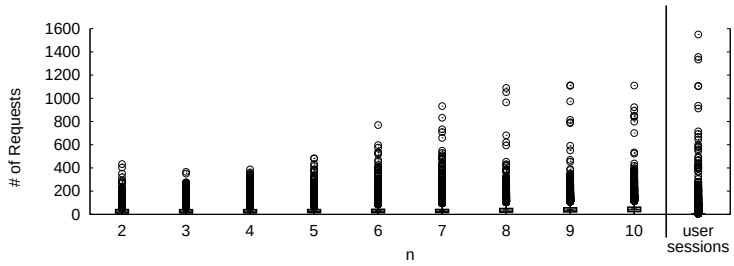


Figure 11. Distribution of Abstract Test Case Length in Terms of Number of Requests Across All Applications, Compared to User Sessions

In summary, in terms of the percentage of each user session's requestType+resource+parameter names sequences of lengths from 1 to 10 that are covered by the generated abstract test cases, the generated abstract test cases using $n \geq 2$ represent user navigation behavior very well in general.

Given these coverage results, a natural follow-up question is how large the generated abstract test cases are. Figure 11 shows the length of the generated abstract test cases compared to the length of the original user sessions. As n increases, the median and mean lengths increase in general, with the exception of n=7 for the median and n=3 for the mean. The generated abstract test cases are similar in length to the original user sessions, although the original user sessions in general have a smaller median and mean and the generated abstract test cases do not have the same magnitude in outliers that the original user sessions have. The increasing lengths of the abstract test cases as n increases may partially explain the increased sequence coverage as n increases shown in Figure 10, but we believe that length has a relatively minor impact.

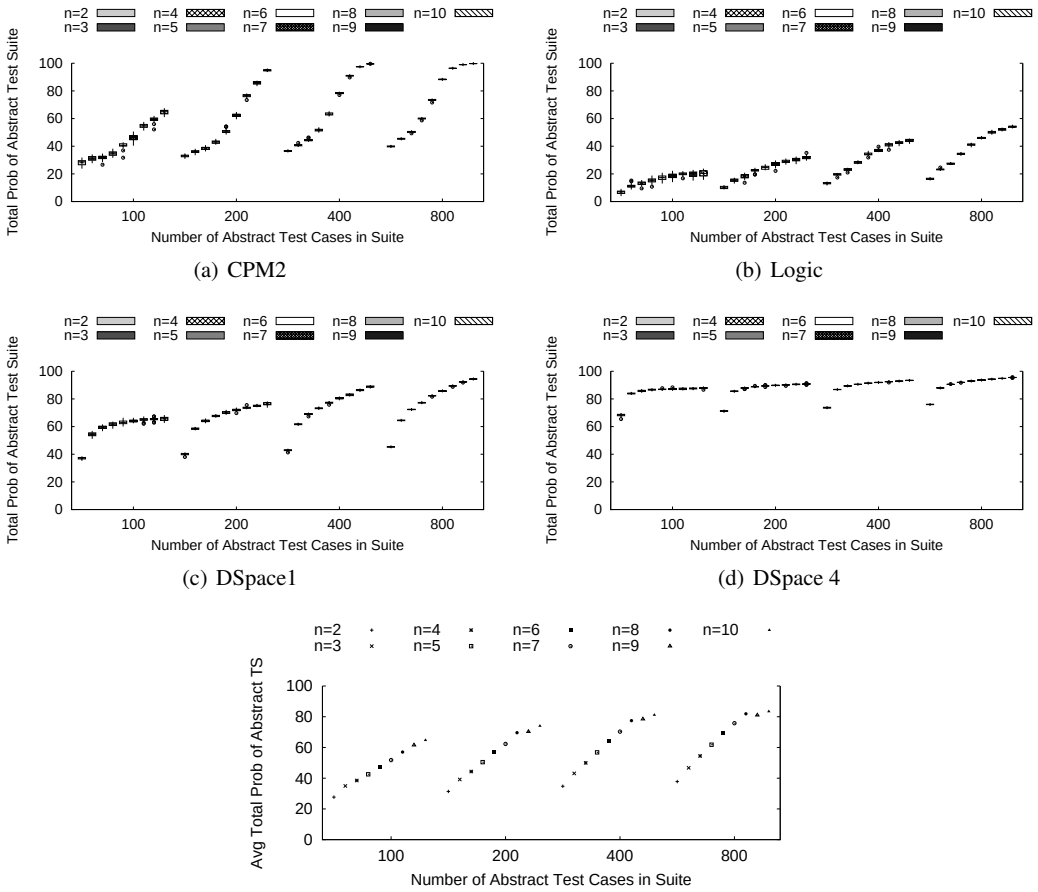
(Q. 4.2.a) Representation of User Sessions: Probability of Abstract Test Case Suites. Figure 12 shows the distribution of the total probabilities of 30 abstract test suites of varying sizes, generated from models using various values of n, for representative applications. For each abstract test case in a suite, we multiplied the probabilities of each traversed model edge used to generate the abstract test case together and then added the probabilities of the test cases together to produce the total probability of the test suite, as in Tonella and Ricca's paper [1]. In general, larger probabilities indicate that the test suite represents the user sessions' common behavior well. The x-axis represents the number of abstract test cases in the suite. The y-axis represents the total probability of the suite, based on the sequences seen in the user sessions. The box represents the middle 50% of the data (the inner quartile range IQR), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The center horizontal line within each box denotes the median, $+$ represents the mean, and $\circ$ represents outliers. The boxes are small, and thus the distinction in shading is hard to see. We always plot n=2 as the left-most distribution and increase to n=10 on the right. Large probabilities indicate that the test suite represents common usage well.

In general, there is not a wide variation in probabilities across the 30 test suites generated from a set of user sessions. This result is not too surprising because we are likely to generate similar or the same highly probable test cases in each suite generated from the same model because of the weighted random walk. Also not surprising is that as number of test cases increases, the probability of the test suite increases. However, even with 800 test cases, we rarely reach a total of 100% probability, which is not surprising given the size and complexity (e.g., cycles) of the navigation graphs.

We also note that as n increases, the probability of the test suite increases. Intuitively, this result makes sense: larger values of n mean we are using more user session sequence history to create the test cases, which means we should generate test cases with higher probabilities. This result also coincides with our previous empirical results: as n increases, the number of states with only one outgoing edge also increases (Table IV), which means that the probability for that edge is 1. Including such an edge in the test case will not decrease the test case's total probability.

It is interesting to note the distinct curve shapes for each application as n increases. As the number of test cases increases, the shape of the curve is similar for each application. If the curves were

DOI: 10.1002/stvr



(a) CPM2

(b) Logic

(c) DSpace1

(d) DSpace 4

(e) Average Across All Abstract Test Suites

Figure 12. Probability of Abstract Test Case Suites

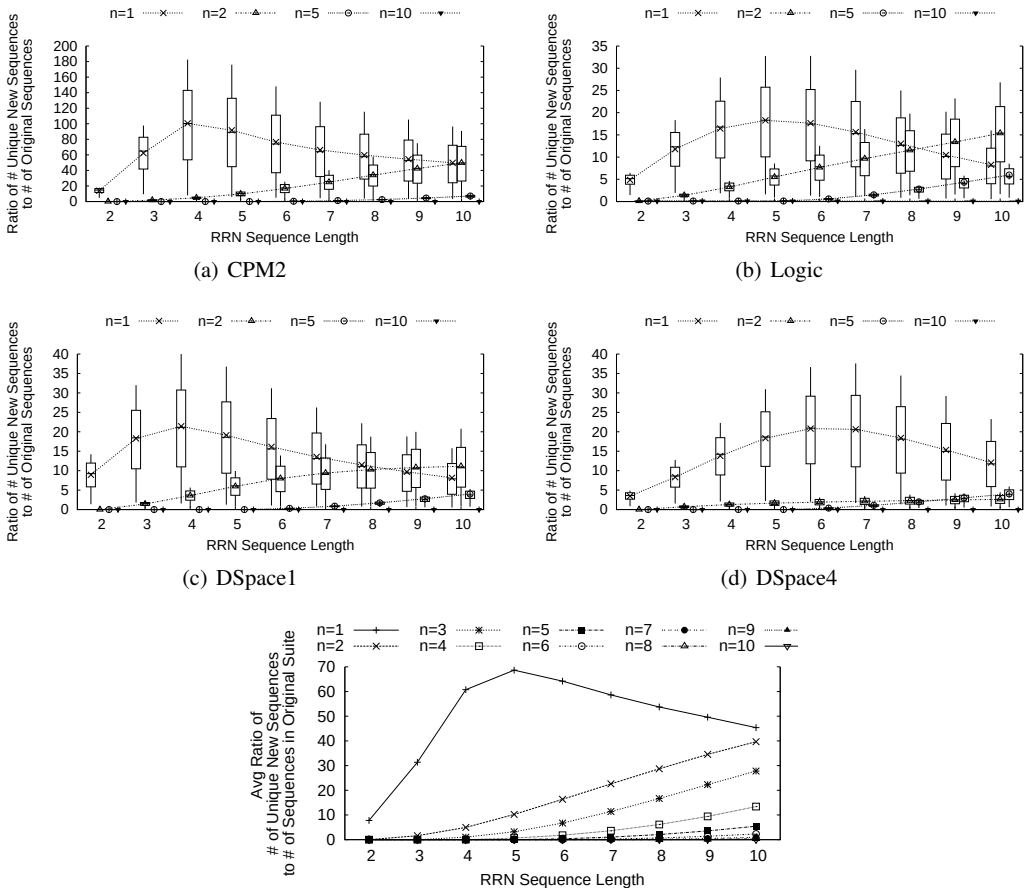
similar across all applications, we may be able to give definitive guidance to testers about how n affects the probability of the test cases. For example, if all curves had the same shape as CPM2 (Figure 12(a)), we would recommend a larger value of n than, say, if all curves had the same shape as DSpace4 (Figure 12(d)).

While not an explicit goal of our experiments, we were curious about the model coverage attained by the abstract test cases, which is related to the total probabilities. We analyzed the abstract test suites' coverage of the navigation model's edges and found that coverage ranged from 40% for DSpace4 when n=2 to 96% for Book when n=9. Not surprisingly, we did not observe any trends as n increases because generation is random and based on probabilities.

An open question remains: are highly probable test cases the most likely to expose faults? The probability metric tends to penalize longer sequences: following an edge often has a probability of less than 1, especially for smaller n. The most probable test cases are usually simply accessing the home page of the site and maybe going to one other page.

(Q. 4.2.b) Potential for Testing New Navigations. Figures 13 shows the ratio of new unique RRN sequences of various lengths that are represented in the generated abstract test cases but *not* in the original user sessions compare to the number of RRN sequences of the same length in the original user sessions. The x-axis represents the length of the RRN sequence. The y-axis represents the ratio of length-x RRN sequences that are represented in the abstract test cases but not in the user sessions compared to the number of length-x RRN sequences in the original set of user sessions. The lines indicate the results for test cases generated from the respective n-gram model.

DOI: 10.1002/stvr



(a) CPM2 (b) Logic

(c) DSpace1 (d) DSpace4

(e) Average Across All Applications' Test Suites

Figure 13. Ratio of the Number of New, Unique RRN Sequences in 500 Abstract Test Cases to the Number of RRN Sequences of the Same Length in the Original User Sessions

Figures 13(a) through 13(d) show representative results for the abstract test cases generated from models of four user sessions sets for n=1, 2, 5, and 10. The box represents the middle 50% of the data (the inner quartile range IQR), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The horizontal line denotes the median and $+$ represents the mean. To improve readability of the graph, we do not show outliers. Figure 13(e) shows the average across all sets of user sessions for $1 \leq n \leq 10$.

As n increases, the ratio of the number of new sequences of all lengths compared to the number of sequences in the original test suite decreases, with a few exceptions, for all sets of user sessions. The maximum number of new sequences is on the order of thousands for all suites of abstract test cases, generated from all sets of user sessions. Thus, even with significant representation of user behavior in terms of sequences of all sizes (Figure 10), the generated abstract test cases also represent a significant number of new, likely navigation paths of various lengths. For example, for CPM2 using $n = 2$ (Figure 13(a)), an average of approximately 40,000 new, unique length-10 sequences—a little over 4 times the number of sequences in the original set of user sessions—are generated. While 40,000 is a large number of new sequences, the sequences are *likely*, based on usage, and 40,000 is much less than 1% of the total number of possible length-10 sequences.

The shapes of the graphs for $n = 1$ and $n \geq 2$ are quite different: there is a distinct bend in the $n = 1$ new sequence curve, often when the sequence length is between 4 and 7, while for $n \geq 2$, the curves are monotonically increasing. While there are more possible sequences of length 10 than length 9, the generated abstract test cases using $n = 1$ do not cover those—likely because the

DOI: 10.1002/stvr

generated $n = 1$ test cases are not sufficiently long because the length was set to 15. The test cases generated using $n \geq 2$ are on average less than 15 requests long, but some test cases are much longer (Figure 11).

In summary, Figures 10, 12, and 13 demonstrate a clear tradeoff: as n increases, the representation of the actual user navigations and the probability of the abstract test cases increase, while the number of new sequences represented by the abstract test cases decreases.

Implication for testing: *A tester could generate abstract test cases until reaching the desired RRN sequence coverage and reduce the abstract test cases, or the tester could analyze the abstract test cases' coverage of the model and create test cases that traverse the uncovered sequences. If a tester wants to generate tests that are more closely aligned in navigation to the user sessions and are, thus, more probable based on usage, then she should choose a larger n. If the tester is interested in more new navigations being tested, then she should choose a smaller n.*

5.4. Threats to Validity

Since we performed our study on thirteen sets of user sessions for five applications, a study with additional sets of user sessions and applications may be necessary to generalize the results. However, we chose applications with different usage characteristics, technologies, implementers, and application characteristics and gathered real user accesses to deployed web applications over a long period of time to improve the chances of seeing different usage patterns and to help reduce the threat to generalizing our results.

Our collected user sessions do not completely cover the code, as discussed in Section 5.1. If we had a more comprehensive set of user sessions, we may see different results. For example, the navigation models would likely show larger model growth—in both RRN and RRNV—as n increases, and the generated abstract test cases would be less likely to cover sequences not seen in the original user sessions, resulting in a clearer indication to the tester for using a smaller n.

For the 1-gram navigation model, we chose for the abstract test cases to contain 15 requests. We could have selected a different number—for example, a larger number of requests would increase the chances of the abstract test cases' representing sequences from the original user sessions—or used an application-specific number of requests for each application. We chose a number that was comparable to the sizes of the user sessions across all applications. The optimal length for abstract test cases generated from a 1-gram navigation model remains an open question.

6. RELATED WORK

Earlier in the paper, we described the research most closely related to this work—namely, navigation models for web applications [1, 3, 4, 5, 6] in Section 2.2.

Deng et al. [4] take a white-box-based approach to testing web applications, focusing on how the database affects the application. We discussed their approach to generating a navigation model in Section 2.2. However, we did not discuss their proposal to use the cyclomatic number to find the minimal number of paths that can generate all possible navigation paths through the application. It is an open question if using the set of minimal paths will result in the most effective testing. For example, the set of minimal paths is not unique—one set may be more representative of users (based on usage data) or may be more likely to expose faults based on other criteria. This is an area of future work.

Harman and Alshahwan [18] analyze changes to a graph—essentially a web application navigation model, although they do not call it that—to determine how to repair user sessions for use in regression testing. Their navigation model is made up of nodes—the pages visited—and edges—links followed to go from one page to another. We did not discuss their navigation model in the background section because the model seems to be a hybrid approach of spidering the application and analyzing the source code and is not the focus of their work. We believe their approach to repairing navigation issues in user sessions for regression testing could be applied to usage-based navigation models as well.

DOI: 10.1002/stvr

Ran et al. [19] seek to test web database applications using a state transition diagram, created from specifications, to model the transitions between "logical" web pages and then generate test sequences from that diagram. The state transition diagram is similar in concept to a navigation model in that is it is used to generate test sequences and then has a separate step to generate data, but the transitions also include form inputs and the application's database state.

Brooks and Memon developed a usage-based navigation model for Graphical User Interfaces (GUI) testing [20]. A variation of this model may be applicable to web testing, although GUI applications have issues—such as infeasible paths—that are not issues for web applications.

Herbold et al. [12] suggest a model for usage-based testing of *any* type of event-driven software, which includes web applications, GUIs, and embedded software. We believe our work fits into their model and is a specific implementation of their model for web applications. The authors propose (but do not implement and evaluate) using variable-order Markov models instead of the fixed-length amount of history we explored in this paper. Implementing and evaluating a variable-order model is another area of future work.

Approaches based on statically modeling web applications [21, 22, 23, 24] are challenged by the common dynamic features of web applications, particularly the dynamically generated pages and non-traditional control flow. Approaches based on modeling web applications with finite-state machines (FSMs) and using coverage criteria based on FSM test sequences are not meant to represent invalid inputs and suffer from the state explosion problem, which has been partially addressed by constraining inputs [25].

Halfond and Orso [26] mention that it would be interesting to combine their approach to testing web application interfaces with user-session-based testing. Their technique is based on static analysis of individual Java servlets and 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. This approach does not test sequences of requests or browser-based inputs but appears to be complementary to usage model-based test case generation.

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 [27, 28]. Concrete and symbolic execution and constraint solving are combined to automatically and iteratively create new input values to explore additional control flow paths through a PHP script. We believe our insights can be applied to concolic testing-based approaches as they can be viewed as building a partial control model through user simulation of the application's menus and buttons.

Alshahwan and Harman [29] introduced a search-based approach to automatically generating test cases for web applications, with the goal of maximizing branch coverage of the application under test. Static analysis collects static information that is used by the search-based algorithm with both static and dynamic seeding. Targeting PHP applications, they demonstrated good fault-finding ability on six large applications. However, the approach is not fully automated, requiring some information to be manually provided.

Some groups propose testing of web applications based on tests directed at the database. For example, Emmi, Majumdar, and Sen [30] apply symbolic execution to solve constraints on SQL queries and then uses this information to generate test cases for database-based web applications. Deng et al. [4] take a white-box-based approach to testing web applications, focusing on how the database affects the application. The authors create a model of the application by extracting URL links from the source code.

Several teams have developed techniques and tools for testing directed at exposing attacks on the web application, including bypass testing to expose threats from bypassing input validations coded in the client-side interface [31], identifying and countering SQL injection attacks [32], and automated parameter mismatch identification [33].

Alshahwan et al. [34] propose several metrics to predict if an automatic crawler is sufficient to achieve high coverage of a web application's page structure. The metrics help to indicate when a random-based automated test data generation strategy will not provide complete coverage of the application. Their evaluation showed that the metrics can be reasonable indicators of crawlability.

DOI: 10.1002/stvr

7. SUMMARY AND GUIDANCE TO TESTERS

As web applications continue to grow in use and complexity, automatic test case generation becomes more important. In this paper, we present empirical evidence supporting specific design decisions in configuring usage-based navigation models for automatic generation of abstract test cases, to be used as a basis for executable test cases. In addition to substantiating some intuitive tradeoffs and researchers' beliefs, our results suggest how the abstract test case suite can be tailored toward representing actual user behavior or potential new navigations.

Specifically, our results showed the following: a tester can tune the amount of history (e.g., n) based on testing and representation goals. A tester should use resource+parameter names to represent user requests, which generally balances scalability and representation requirements, but may want to consider including parameter values in the representation if using a larger n because the additional increase in model size is not as large. When building the navigation model, a tester should arrange user sessions in decreasing length order to observe growth stability trends better and stop adding user sessions when growth stability goals are met. Building models using larger values of n requires more user sessions to reach growth stability goals. However, exponential model growth with a large number of user sessions when representing requests using resource+parameter names is not a concern because of the redundancy in user sessions, and, thus, a tester may want to use more user sessions to provide more accurate estimates of users' transition probabilities. Usage information is important to model because user behavior does not follow exactly a statically determined model and, when given several options, users tend not to choose between the options equally. In terms of the abstract test cases, generating 500 test cases resulted in good representation of the user behavior seen in the user sessions as well as likely user behavior not seen in the user sessions in our experiments; however, determining the number of abstract test cases to generate remains an open question. The abstract test cases also show a tradeoff in choosing a value for n: larger values of n result in better representation of actual user behavior but less testing of user behavior not seen in the user sessions.

To summarize the tradeoffs in choosing a value of n: larger values of n result in larger models (although the increase in model size decreases as n increases) that can be used to generate larger, more probable test cases that represent user behavior, while also testing additional, likely user behavior. More user sessions are required to generate stable navigation models than using a smaller value for n.

Armed with knowledge of how to best generate abstract test cases, we are currently investigating potential data models for generating parameter values to be integrated with the abstract test cases to construct effective executable test cases. We also plan to evaluate the impact of the abstract test cases on data models and the effectiveness of the resulting executable test suites in fault detection and code coverage in publicly deployed web applications.

ACKNOWLEDGEMENTS

We thank Lloyd Greenwald of Sant et al. [5] for clarifying their representation of user requests and implementation. We thank Natallia Robinson, who developed the Logic application. We thank Megan Fulcher for her help in statistical analysis of edge probabilities. Finally, we thank Emily Hill for her help in analyzing the edges' probabilities and feedback on earlier versions of the paper.

REFERENCES

1. Tonella P, Ricca F. Statistical testing of web applications. *Journal of Software Maintenance and Evolution* 2004; **16**(1-2):103–127.
2. Halle S, Ettema T, Bunch C, Bultan T. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. *International Conference on Automated Software Engineering (ASE)*, 2010.
3. Kallepalli C, Tian J. Measuring and modeling usage and reliability for statistical web testing. *Transactions on Software Engineering* 2001; **27**(11):1023–1036.
4. Deng Y, Frankl P, Wang J. Testing web database applications. *SIGSOFT Software Engineering Notes* Sep 2004; **29**(5):1–10.

DOI: 10.1002/stvr

5. Sant J, Souter A, Greenwald L. An exploration of statistical models of automated test case generation. *International Workshop on Dynamic Analysis (WODA)*, 2005.
6. Wang W, Lei Y, Sampath S, Kacker R, Kuhn R, Lawrence J. A combinatorial approach to building navigation graphs for dynamic web applications. *International Conference on Software Maintenance (ICSM)*, 2009.
7. Elbaum S, Rothermel G, Karre S, Fisher II M. Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering* 2005; **31**(3).
8. Sprenkle S, Pollock L, Simko L. A study of usage-based navigation models and generated abstract test cases for web applications. *International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2011; 230–239.
9. Sprenkle S, Gibson E, Sampath S, Pollock L. A case study of automatically creating test suites from web application field data. *Workshop on Testing, Analysis, and Verification of Web Services and Applications (TAVWEB)*, 2006.
10. pydot. http://code.google.com/p/pydot/ 2012.
11. Graphviz DOT language. http://www.graphviz.org/ 2012.
12. Herbold S, Grabowski J, Waack S. A Model for Usage-based Testing of Event-driven Software. *3rd International Workshop on Model-Based Verification & Validation From Research to Practice*, IEEE Computer Society, 2011.
13. Open source web applications with source code. http://www.gotocode.com 2003.
14. DSpace Federation. http://www.dspace.org/ 2012.
15. Cobertura. http://cobertura.sourceforge.net/ 2012.
16. Sampath S, Sprenkle S, Gibson E, Pollock L, Greenwald AS. Applying concept analysis to user-session-based testing of web applications. *Transactions on Software Engineering* October 2007; **33**(10):643–658.
17. Sampath S, Sprenkle S, Gibson E, Pollock L. Web application testing with customized test requirements—an experimental comparison study. *International Symposium on Software Reliability Engineering (ISSRE)*, 2006.
18. Harman M, Alshahwan N. Automated session data repair for web application regression testing. *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, IEEE Computer Society, 2008; 298–307.
19. Ran L, Dyreson C, Andrews A, Bryce R, Mallery C. Building test cases and oracles to automate the testing of web database applications. *Information and Software Technology* Feb 2009; **51**(2):460–477.
20. Brooks PA, Memon AM. Automated GUI testing guided by usage profiles. *International Conference on Automated Software Engineering (ASE)*, 2007.
21. Alalfi MH, Cordy JR, Dean TR. Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification, and Reliability* 2009; **19**(4):265–296.
22. Di Lucca G, Fasolino A, Faralli F, Carlini U. Testing web applications. *International Conference on Software Maintenance (ICSM)*, 2002.
23. Liu CH, C KD, Hsia P, Hsu CT. Structural testing of web applications. *International Symposium on Software Reliability Engineering (ISSRE)*, 2000.
24. Ricca F, Tonella P. Analysis and testing of web applications. *International Conference on Software Engineering (ICSE)*, 2001.
25. Andrews AA, Offutt J, Dyreson C, Mallery CJ, Jerath K, Alexander R. Scalability issues with using FSMs to test web applications. *Information and Software Technology* 2009; .
26. Halfond WGJ, Orso A. Improving test case generation for web applications using automated interface discovery. *Foundations of Software Engineering (FSE)*, 2007.
27. Artzi S, Kieżun A, Dolby J, Tip F, Dig D, Paradkar A, Ernst MD. Finding bugs in dynamic web applications. *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.
28. Wassermann G, Yu D, Chander A, Dhurjati D, Inamura H, Su Z. Dynamic test input generation for web applications. *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.
29. Alshahwan N, Harman M. Automated web application testing using search based software engineering. *International Conference on Automated Software Engineering (ASE)*, 2011; 3 –12.
30. Emmi M, Majumdar R, Sen K. Dynamic test input generation for database applications. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2007; 151–162.
31. Offutt J, Wu Y, Du X, Huang H. Bypass testing of web applications. *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society, 2004; 187–197.
32. Halfond W, Orso A, Manolios P. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2006; 175–185.
33. Halfond W, Orso A. Automated identification of parameter mismatches in web applications. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2008; 181–191.
34. Alshahwan N, Harman M, Marchetto A, Tiella R, Tonella P. Crawlability metrics for web applications. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE Computer Society, 2012; 151–160.

DOI: 10.1002/stvr