LEVERAGING PARAMETER AND RESOURCE NAMING CONVENTIONS TO IMPROVE TEST SUITE ADHERENCE TO PERSISTENT STATE CONDITIONS

by

Johanna Goergen

2016

TABLE OF CONTENTS

LIST OF TABLES . . . . . **v**
LIST OF FIGURES . . . . . **vi**
ABSTRACT . . . . . **viii**

Chapter

1 INTRODUCTION **1**

2 BACKGROUND **3**

2.1 Web Applications 3
2.2 Web Application Testing 5

2.2.1 Automated Web Application Testing 6

3 THE PROBLEM: INCOHERENT SELECTION OF REQUEST PARAMETERS IN TEST CASES **8**

4 EXPLORATORY STUDIES **11**

4.1 Resource Naming Conventions Study 11

4.1.1 Methodology 11
4.1.2 Results and Evaluation 12

4.2 Parameter Naming Conventions Case Studies 13

4.2.1 Subject Applications 14
4.2.2 Methodology 14
4.2.3 Results and Evaluation 16

4.2.3.1 Logic 16
4.2.3.2 ORPM 18

4.2.3.3 WebCollab 18

4.3 Conclusions from Exploratory Studies 19

5 APPROACH: CLASSIFICATION EXTRACTOR AND PSE-MATCHING ALGORITHM 21

5.1 Classification Extractor 23
5.2 PSE-Matching Algorithm 26

5.2.1 Example of PSE-Matching Algorithm 27
5.2.2 PSE-Matching Algorithm Rules 30

6 EXPERIMENT, RESULTS, AND ANALYSIS 36

6.1 Research Questions 36
6.2 Experiment Methodology 36

6.2.1 N-Gram Simple Model 37
6.2.2 Parameter Interaction Model 39

6.3 Metrics 40
6.4 Results and Analysis 42

6.4.1 Improvements in Line and Branch Coverage 42
6.4.2 Analysis of Coverage Data 45
6.4.3 Analysis of High and Low -Improvement Resources 48
6.4.4 Time Complexity of Classification Extraction and PSE-Matching Algorithms 50

6.5 Threats to Validity 51
6.6 Summary of Analysis and Observations 51

7 CONTRIBUTIONS AND FUTURE WORK 53

7.1 Contributions 53
7.2 Future Work 54

BIBLIOGRAPHY 55

LIST OF TABLES

4.1 List of ID-Indicators 14

4.2 ID-Parameter Data from Three Subject Applications 16

5.1 Terminology for the Classification Extractor and the PSE-Matching Algorithm 24

6.1 Number of requests per resource in 50-case WebCollab test suite . 46

6.2 Significant data from high-improvement and low-improvement resources from Logic's 30 test suites 48

6.3 Time requirements for the Classification Extractor 50

6.4 Time requirements for the PSE-Matching Algorithm 50

LIST OF FIGURES

2.1 Components of a web application, including session and persistent state . . . 3

2.2 Components of HTTP requests referenced in this paper . . . 4

2.3 Web aplication components, including requests and responses . . . 4

3.1 Requests ordered in test suite without respect to persistent state requirements . . . 9

3.2 Parameter requiring value from persistent state . . . 9

3.3 Parameter requiring unique value . . . 10

4.1 PSE in Resource Name Matches Parameter Name . . . 16

4.2 An ID-Parameter for a Course PSE . . . 17

4.3 Another ID-Parameter for a Course PSE . . . 17

4.4 Ambiguous ID-Parameter in ORPM . . . 18

4.5 Action Parameter Example . . . 19

5.1 Input and output for the Classification Extractor and the PSE-Matching Algorithm . . . 21

5.2 Classification Extractor example on Logic request . . . 25

5.3 Identifying key elements of a request . . . 28

5.4 Inferring the request's effect on entities in persistent state . . . 28

5.5 Selecting eligible parameter values for the request . . . 29

5.6 Updating the Persistent-State Map upon acquiring an eligible parameter set 29

5.7 Create-classified request creating an entity already in persistent state 30

5.8 Read-classified request accessing nonexistent entity in persistent state 31

5.9 Update-classified request in which no ID-Parameter for the Request PSE exists 32

5.10 Delete-classified request in which no ID-Parameter for the Request PSE exists 34

6.1 Example of conditional probabilities for selecting parameter sets . 39

6.2 Function for coverage improvement threshold 41

6.3 Average line coverage of 10 simple test suites v. PS-aware test suites with Logic as subject application 43

6.4 Average branch coverage of 10 simple test suites v. PS-aware test suites with Logic as subject application 43

6.5 Line coverage of most improved Simple test suite v. PS-Aware test suite by size with Logic as subject application 44

6.6 Branch coverage of most improved Simple test suite v. PS-Aware test suite by size with Logic as subject application 45

6.7 Line Coverage of Simple test suites v. PS-Aware test suites by size with WebCollab as a subject application 46

ABSTRACT

With the prevalence of web applications increasing daily in various aspects of modern life, the need for cost-effective, efficient, and thorough web application testing is now greater than ever. One approach to web application testing is automatic test suite generation based on real user sessions. This approach is promising, but tends to leave a considerable amount of web application functionality untested. It remains overall ineffective due to most automatically generated test suites' lack of adherence to the persistent state of the application under test, or the parts of the application's state that are stored in a data store external to the application itself. This thesis explores the possibillity of leveraging the resource and parameter naming conventions of web applications to automatically predict the actions test cases will perform on data in persistent application state. I propose an algorithm that creates these predictions and another that leverages these predictions to improve test suites to more closely adhere to the conditions of data in persistent state. These improvements are achieved through optimization of request parameter selection. I perform experiments to determine the success of the two algorithms and I propose suggestions for improvements as well as suggestions for future work.

Chapter 1

INTRODUCTION

A web application is a software application whose functionality can be accessed by users over the Internet via web browsers. As web applications take on more and more vital, sensitive responsibilities such as banking, tax, and health services, it is increasingly critical that web applications are well tested and maintained before they are deployed to the public and with every subsequent update or change. For software in general, the testing process "accounts for approximately 50% of the cost of software system development" [6]. Testing of web applications in particular can be extremely expensive and time-consuming due to the dynamic nature of the content they generate.

Much of the content displayed by web applications is dynamically generated by the application code as the user navigates the application. Some of this dynamic content can also be determined by data acquired from a database or other data store outside the application itself. For example, consider an online shopping application in which a user saves his shopping cart and logs out. If that user is able to access the items in his shopping cart and purchase them upon his next login, the dynamically generated content of his shopping cart must have been stored in a database during his first session and accessed again during his second. Ensuring the correctness of even this simple shopping transaction requires a great deal of testing. Specifically due to the incredibly large domain of input possible for web applications, it is often infeasible to test all possible web application functionality manually.

As a result, a great deal of web application testing is automated and many approaches to automated web application testing exist. A common current approach to automating web application testing is test suite generation based on user sessions [4, 5, 8, 10]. Although this approach to automated testing is very promising, it leaves

room for improvement in effectiveness due to its lack of adherence to requirements imposed by data outside of subject application code, such as data stored in databases. My objective is to contribute an approach to creating more effective usage-based web application test suites based on predicting the content of the subject application's external data store(s) throughout testing.

The contributions of my work are:

1. Language-based techniques for determining HTTP request's implications for persistent application state
2. Dynamic prediction and tracking of elements in persistent application state throughout traversal of a test suite
3. Algorithm to apply persistent state predictions and create a persistent-state aware test suite
4. Persistent-state aware test suite improves coverage for applications written in two different programming languages

The organization of this thesis is as follows: in Chapter 2, I elaborate upon the definition of a web application and the role persistent state plays in the testing process. In Chapter 3, I more specifically outline the shortcomings of current automatic testing methods which I hope to improve upon. Chapter 4 details the two exploratory studies I performed in order to inform the direction of my further approach. In Chapter 5, I provide a hypothesis and approach based on the findings of these exploratory studies. Chapter 6 reveals the results of my approach and offers an analysis of the results, also detailing various algorithms developed to provide a more comprehensive analysis of test suite success. Chapter 7 offers conclusions and recommendations for future work.

Chapter 2

BACKGROUND

2.1 Web Applications

A web application is a system of web pages and other resources on a web server with which a user can interact via a web browser. A web application dynamically determines the content rendered by the client browser based on information received from the user and/or by accessing *session state* and *persistent state*. *Session state* refers to the data created on the server throughout a *user session*, which is a series of interactions between one user and the application, delimited by periods of inactivity greater than some certain amount of time by that user. Throughout a user session, the data created by the user as well as that created in previous sessions and stored in *persistent state*, or a data store outside the application code, is used to dynamically determine content seen by the user. Figure 2.1 shows the components of a web application, including session and persistent state.

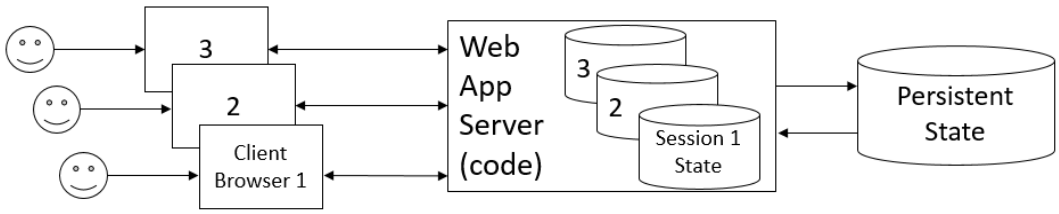

Figure 2.1: Components of a web application, including session and persistent state

To specify certain actions such as navigation through the web application or submission of forms to the application, the user interacts with the web page in the client browser which in turn sends a *request* to the application server. By request, I refer to an *HTTP (HyperText Transfer Protocol) request*.

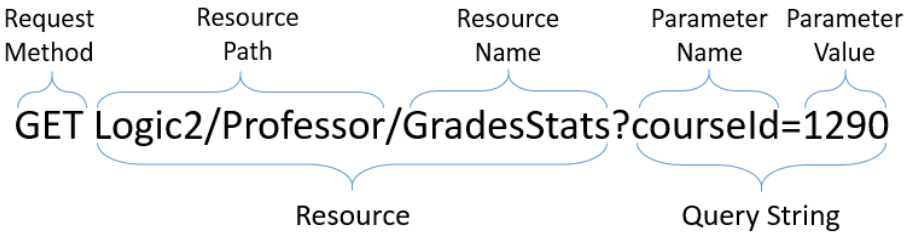

Figure 2.2: Components of HTTP requests referenced in this paper

An *HTTP request* is a request for a resource belonging to the web application. Figure 2.2 identifies the components of a request I will be referencing in this thesis; however, HTTP requests often contain much more information than displayed. An HTTP request is sent by the user's browser to the web application server, and it contains information to allow the server to respond with requested resource to the correct place, in the correct form. The request consists at least of a request method– usually GET or POST– as well as a resource and zero or more parameters consisting of name/value pairs. The web application server then returns a response, whose body is typically an *HTML (HyperText Markup Lanuage)* document, which the client browser renders. The HTML document often contains elements that the user can interact with to then send further HTTP requests. The components of a web application used with a browser and shared data store are shown in Figure 2.3.

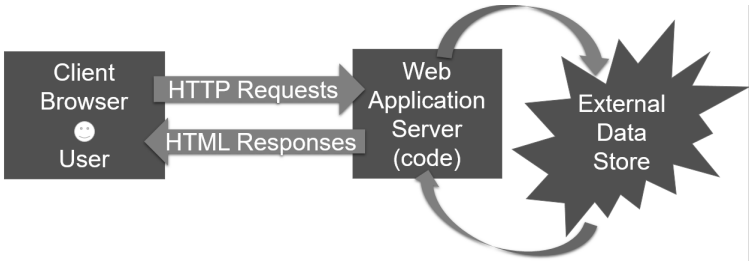

Figure 2.3: Web aplication components, including requests and responses

If a particular response has content determined by the application code based

on variable input, then this response is *dynamic*. For example, a response is dynamic if the parameters passed in a request are used by the application code to determine the exact content returned by the server. Suppose an admin and a general user of some web application both log in to the web application from the same login page. This login request may result in the rendering of a different HTML page for the admin than for the general user if the admin can access different functionality than the general user. This means the response was dynamic because application code must have determined which user was an admin and which was a general user based on their usernames and passwords. Other examples of dynamic responses are HTML pages displayed as the result of a specific search query, where that search query string is the parameter determining the content seen by the user.

Dynamic responses are most often determined by how the request accesses and changes an application's persistent state. Persistent state is the web application's state that is not a part of the application code itself but is instead stored in some outside data store such as a database or text file. In Figure 3, the shared data store represents persistent state. *Persistent state* is state maintained across sessions that allows a web application user to access, modify, or remove data created in previous user sessions or even in the user sessions of other users with the same application. For example, when a user account is created in one session, the new user's username and password will likely be stored in a database and required at this user's next login attempt. Such use of outside data stores is the only way to create a continuity and coherence between different user sessions, allowing the user account created in one session to be accessible by the new user in his subsequent attempt to log into the web application.

2.2 Web Application Testing

Functionality testing of web applications seeks to exercise a web application's functionality by executing as much application code as possible, in terms of both code branches and lines, to reveal any faults in the code. Upon executing the application code, many web application testing services compare the HTML responses displayed by

the browser with the expected HTML responses, ensuring application code correctness for as many cases as possible. While this exhaustion of application functionality may be easily achievable manually for small applications with few resources and a simple navigational model, it becomes incredibly time and cost ineffective to manually exercise all application functionality for large, highly dynamic web applications. For this reason, web application testing has become a predominantly automated process.

2.2.1 Automated Web Application Testing

To decrease the time and resource costs necessary to test web applications, a common approach is automation of testing. Approaches to automatic testing use many different models to generate *test suites*. A *test suite* is typically an ordered series of *test cases* meant to model user sessions. Each *test case* consists of a series of requests, which provide the information specified in Figure 2.2. Although HTTP requests contain much more than just the resource, method, and parameters, these three components are the three necessary to create a model interaction with a web application when testing because the session and cookie information can be automatically handled by a test case execution tool.

Today, one common method for automated software testing is utilizing user access logs to create a usage-based testing model [4, 5, 8, 10]. To create usage-based test suites, it is necessary to mine actual accesses to a subject web application. These access logs are then partitioned into user sessions. From these user sessions, any number of algorithms can be used to develop *abstract test suites*, ranging from creating Markov usage models to ordering requests at random. An *abstract test suite* is a series of *abstract test cases* that each consist of requests without parameter values specified. Lack of parameter specification makes the test suite abstract and allows the automated testing algorithm the possibility to apply additional models in choosing parameter values.

Usage-based testing offers benefits not offered by various white-box web testing methods. For example, reliance on usage-based statistical models allows test suites to

more exhaustively test the most common usage patterns (i.e. the functionalities of the application that will likely see the most user traffic) while not spending more time or resources than necessary on uncommon usage patterns (i.e. functionalities that users of the web application will not likely use often) [5]. Researchers have employed many combinations of statistical and usage-based testing models to achieve high-coverage testing algorithms, such as Sant et al's *Simple Model*, which creates usage-based Markov chains, giving the likelihood of any one request based on the one or two requests preceding it [9]. The *Simple Model* creates a series of abstract test cases in this fashion (resulting in an abstract test suite), then specifies parameter values for the requests in those abstract test cases by referencing another Markov assumption model for selection of parameter values.

Chapter 3

THE PROBLEM: INCOHERENT SELECTION OF REQUEST PARAMETERS IN TEST CASES

Since I consider a test suite to be a series of test cases meant to model user sessions, an ideal test suite retains a level of logical coherence throughout the traversal of its individual cases in order. In other words, the test cases meant to test non-error code within a suite should create information that can be successfully stored and accessed, as well as change or delete existing information successfully. When applications rely heavily on external data stores, however, it is difficult to automatically generate test suites that correctly model interactions with these external data stores. This is because it is not always clear which application resources access or create external data. Certain requests may only succeed if particular information exists within the external data store, and others only if the data store is empty. Most algorithms to automatically generate test suites are unable to take these requirements imposed by persistent application state into account and as a result, the test suites they generate are often unable to achieve high code coverage. This inability of automatic test suite generating algorithms to adhere to the conditions of persistent state often results in the following defects in test suite coherence:

1. **Test cases ordering creates logical inconsistencies.** Some test cases require prerequisite actions to occur before they are able to execute non-error code successfully. For example, in testing a human resources management application, if a series of test cases modeling employee user sessions are executed before a test case modeling an admin user session adds employees to the application's database, those initial employee test cases will repeatedly hit the same error code, likely not executing any code beyond the login page. As in this example, it is common that the successful execution of code by one test case relies on the previous execution of some other code by another test case. Figure 3.1 provides an example of

illogical test case ordering. Without the ability to interact with an application's outside data stores, a test suite generating algorithm cannot easily know where such dependencies lie, and therefore will fall short in the way it orders test cases. This results in excessive coverage of error code and sparse coverage of the rest of the application code.

Within Test Case #1:

POST Webcollab/users.php --post-data="&action=delete&**userid=4**"

Within Test Case #2:

POST Webcollab/users.php --post-data="&action=create&**userid=4**&username=jgoergen&password=123"

Figure 3.1: Requests ordered in test suite without respect to persistent state requirements

In Figure 3.1, the first test case attempts to delete a user with ID 4, but this user is not created until the second test case. The illogical ordering will result in coverage of error code instead of code that performs deletion of the user.

2. **Test cases contain requests which refer to data in persistent state.** Requests that are intended to update, delete, or read data often contain parameters that specify the data to be accessed. Any data stored in persistent state is likely accessed by either an ID string or some combination of characteristics (e.g., a database query specifying one or more attribute), which are specified in some way by the user's interaction with the web application then sent via request. Many approaches to automated test suite generation use some degree of randomness in choosing the parameter values in a request [9]. However, if requests attempting to refer to specific data in persistent state contain randomly selected parameter values, it is unlikely that they will be able to successfully create or access that data. Most often, the parameters of the request will refer to nonexistent data, which will once again result in repeated coverage of error code. In Figure 3.2, I provide a request that requires the existence of a certain entity in persistent state to succeed.

GET Webcollab/tasks.php?action=show&**taskid=47**

Figure 3.2: Parameter requiring value from persistent state

The request shown in Figure 3.2 seeks to access the task with ID 47. This will result in error code if there is no record in the application's persistent state about a task with ID 47.

3. **Test cases contain requests that require new data.** Requests that are intended to create new information to store in persistent state often contain parameters that specify new data (e.g., usernames or IDs not already stored in persistent state) to be saved. As in Case 2, it is less likely that the requests will contain new parameter values when these values are randomized. Figure 3.3 provides an example of a request that requires a unique parameter to succeed in performing its desired functionality. The request seeks to create a new student and add the student's information to its persistent state. The value belonging to the username parameter must be unique (i.e. cannot already exist within the application's state) for this request to succeed. Although it is necessary to test situations in which a user attempts to create duplicate data to assure that the web application does not allow this, it is not ideal to cover this exception-handling code excessively. Thus, test cases with persistent state having requests of this type will cover error code too heavily, as in the cases above.

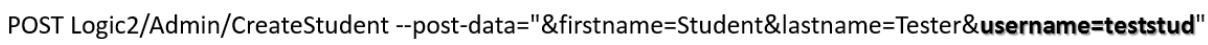

Figure 3.3: Parameter requiring unique value

Considering the above limitations, the inability of test suite generators to access a subject application's persistent state appears to severely limit the logical coherence of parameter values throughout test suites. Therefore, the research I present aims to improve the logical continuity of automatically generated test suites by specifically improving the parameter values selected within the test suites requests. I hypothesize that it is possible to use resource and parameter naming conventions to predict dependencies created and changed throughout a test suite in order to make these improvements in parameter value selection.

Chapter 4

EXPLORATORY STUDIES

To provide insight about how parameter values within a test suite's requests might be automatically assigned within the confines of persistent state requirements, I conducted two exploratory studies.

4.1 Resource Naming Conventions Study

This study explores the hypothesis that resource naming conventions used by web developers might allow testing algorithms to infer persistent state information about the requests within a test suite. The ability to classify HTTP requests with regards to the actions they perform on items in a shared data store might allow us to keep track of this data while building test suites.

4.1.1 Methodology

I began by acquiring the Firefox histories of two users and mining 1,980 resource names belonging to 74 well-known web applications (i.e. Amazon, KAYAK, etc.) from the histories. I evaluated these resources for resource naming conventions and patterns manually, searching especially for patterns that indicated dependencies among resources. After manual evaluation of the resource names, it was possible to produce a list of words that establish a resource as one that is likely to affect an application's persistent state as a Create, Read, Update, or Delete (CRUD) operation. I call these words *Persistent State Indicator Words (PSIW)*. More specifically, for a word to be classified as a PSIW word, it needs to meet the following specifications:

1. **Has obvious CRUD classification by common English language standards:** For example, the word "remove" is a Delete-classified PSIW, as it obviously fits this one classification and no others. Similarly, "submit" is Create-classified, as it clearly indicates the intention to create new data for storage.

2. **Appears in resource names belonging to more than one application:** Some web applications might feature a certain word heavily within its resource names and each request utilizing this word might affect persistent state in the same way; however, this word might be an indicator specific to this application only. If a word did not appear in at least two of the web applications in the study, there was not a strong enough case for classifying it as a PSIW. For example, the word "results" was identified in resource names from 8 different applications in the study, each having a likely Read-classification. Therefore, "results" became an R-classified PSIW.

3. **Has same CRUD classification across all resource names:** This requirement ensures that not only is a CRUD-classified word common but also consistent in its implications for persistent state. If a word appears in resources from many applications but these resources perform various types of CRUD operations, the word's presence does not indicate one definitive classification strongly enough to consider it a PSIW. In the "results" example mentioned above, the fact that each of the resources containing "results" had a Read-classification suggests that presence of the word "results" has unambiguous implications in terms of persistent state.

4.1.2 Results and Evaluation

Applying a mixture of manual and script-based means to compile a conclusive list of Persistent State Indicator Words resulted in a list of 49 PSIWs with 29 Reads, 11 Creates, 6 Updates, and 3 Deletes. From this PSIW list, it was possible to categorize a list of 111 Create, 596 Read, 73 Update, and 7 Delete resources out of the original 1,980 resource names from Firefox history. These results indicate that there is some universality in naming conventions when it comes to the use of CRUD words in resource names. It indicates also that there will generally be more resources devoted to simply accessing objects in a session or persistent state than removing, modifying, or creating them, as expected. Upon analysis of the PSIW list alongside the original collection of resource names, an evident question arose: if a PSIW indicates the execution of some CRUD action, what object is this CRUD action performed on? It seems likely that

this information is also derivable from resource names. Creating, reading, updating, and deleting are actions that take direct objects, and the idea that direct objects too should be determinable via analysis of resource names logically follows. Further manual analysis of the mined resource names indicated that indeed, a 72% majority of the 787 CRUD-classified resources had both a PSIW and some noun in their names, which could often be identified as the likely direct object of that CRUD action. In CRUD-classified requests whose resource name can be split to contain one clear object alongside a PSIW (as determined by a splitting algorithm [1]), this object (usually a noun) will be referred to as the request's Persistent State Element. A *Persistent State Element (PSE)* is an object whose state is likely persistent, or an object whose information is likely stored in an external data store.

4.2 Parameter Naming Conventions Case Studies

As explained in Chapter 3, requests that access existing entities in persistent state will often rely upon request parameters to specify the particular objects on which they act. The idea that it may be possible to logically connect a PSE to the parameters used to identify it within a request brought me to the next exploratory study, which focuses on parameter-naming conventions. The Parameter Naming Conventions Study sought to identify patterns in request parameter names, especially linguistic indicators of relationships between parameter names and entities in persistent state. The study of parameter naming conventions was conducive to a case study instead of an analysis of a large dataset (as in the Resource Naming study), due to the fact that access to and some knowledge of subject application code were necessary to perform these studies. After finding requests to be CRUD-classifiable, I wondered if the parameter names in these requests might help identify the PSEs on which the requests act. I hypothesize that using parameter-naming conventions will allow automatic identification of parameters whose values are likely unique ID strings within requests. I will refer to these parameters as *ID-Parameters*. Furthermore, it will be possible to predict the PSEs represented by these unique ID strings.

4.2.1 Subject Applications

The following subject applications were used in this case study:

1. **Logic:** Logic is a Java web application used by Washington and Lee logic courses, with which professors can administer quizzes to students and keep track of students' grades. It is built with Java EE technology (i.e. servlets and JSPs), JavaScript client-side functionality, and PostgreSQL as its database backend. The web application contains 20,965 lines of code.

2. **Online Rental Property Manager (ORPM):** ORPM [7] is a PHP web application by which landlords can track current and past properties, tenants, leases, applications for leases, and lease applicants as well as interact with their tenants and applicants via email. Its back end is written in PHP with support for a MySQL database, and its user interface features components written in JavaScript. It contains 23,231 lines of code.

3. **WebCollab:** WebCollab [11] is a PHP web application that allows an admin to create groups within which multiple users can create, assign, and track projects and tasks. It supports either MySQL or PostgreSQL as its database backend and also uses JavaScript in its frontend without a framework. It contains 29,332 lines of code.

4.2.2 Methodology

From experience with web application development and use of web applications and databases, I compiled a short list of strings I will refer to as *ID-indicators*– strings that, when found within a parameter name, imply the corresponding parameter value to be a unique identifier key used to locate an item within persistent state.

ID-Indicators
ID
Key
Username

Table 4.1: List of ID-Indicators

To provide evidence for the claim that finding ID indicators in parameter names implies unique IDs as values, I gathered three key data points from each subject application. First, I collected usage data from the three applications in the form of

request logs. The usage data for the Logic application was genuine usage data from two semesters of an introductory logic course. The usage data for ORPM and WebCollab was a series of sessions I carried out in an effort to use as much of the applications functionalities as possible while accessing persistent state in a realistic fashion.

First Data Point: From the access logs for each application, I compiled the set of all unique request combinations for each application (i.e. all unique combinations of request method, resource name, and parameter names). For example, in the ORPM application, a GET request for `propertiesview.php` can contain the parameter `t` or it can contain the parameters `SelectedID`, `record-added-ok`, `Embedded`, `SortField`, `SortDirection`, `FirstRecord`, `DisplayRecords`, and `SearchString`. Ill refer to these as different request combinations. Additionally, if it were possible to make a POST request for `propertiesview.php` with the same parameters as either of the above request combinations, this would be considered its own unique request combination and added to the set as such.

Second Data Point: From each application's set of unique request combinations, I gathered the number of unique request combinations with parameters containing ID-Indicators. For example, the WebCollab GET request for `tasks.php` asks for the parameter `taskid`, which the splitting algorithm parses into the two words "task" and "ID". The presence of "ID" in the parameter name implies that the corresponding value for this parameter will be a unique identification string. Thus, the second important data point I gathered is the size of the subset of unique request combinations containing at least one parameter with an ID-Indicator.

Third Data Point: The final data point for each application is the size of the subset of unique request combinations from the above subset containing parameters like `taskid` where the parameter name implies not only the existence of an ID string but also gives the name of the object being acted on– in this case "task". In other words, the third data point is the number of unique request combinations per application containing at least one parameter name with an ID-Indicator as well as a likely Persistent-State Element.

4.2.3 Results and Evaluation

	Unique Request Combinations	Requests with ID-Indicators	Requests with ID-Indicators and a PSE
Logic	75	13	7
ORPM	103	60	33
WebCollab	52	28	28

Table 4.2: ID-Parameter Data from Three Subject Applications

Table 4.2 indicates the basic commonality of the ID-Indicators specified by Table 4.1 as well as some evident variety in naming conventions among different applications. With a deeper evaluation of the results for each individual subject application, I seek to explain why the data presented in Table 4.2 has unique and promising implications for the possibility of predicting interactions with persistent state. I offer the following analysis of the results obtained for each of the three applications:

4.2.3.1 Logic

The Table 4.2 data for the Logic application features a comparatively low frequency of ID-indicator parameters within its unique request combinations, which is surprising because Logic relies heavily on interactions with a database. Upon further inspection, Logic features many requests such as Figure 4.1, where unique ID strings are given names that are equivalent to the PSEs they represent.

POST Logic2/Admin/edit**Course**.jsp --post-data="&**course=1**”

Figure 4.1: PSE in Resource Name Matches Parameter Name

In Figure 4.1, the ID for the course is passed as the value for the parameter named `course`. Conveniently, the resource-naming study indicates that “course” is

the PSE of this request based on the result of splitting the resource name into parts. Out of the 17 Logic requests like the one above that use the name of a known PSE as a parameter name, 15 of those single-noun parameters are confirmed to require unique IDs of the specified PSE type as values. Therefore, not all ID-Parameters are identified by ID-Indicators in their parameter names. If a parameter name is the name of a Persistent State Element, the parameter's value is likely the ID for that PSE. Therefore, an ID-Parameter can either have a name equivalent to a known PSE (see Figure 4.2) or a name containing both an ID-Indicator and a PSE (see Figure 4.3).

```
POST Logic2/Professor/CreateQuiz --post-data="&course=1&quiz_title=Graded+Quiz"
```

Figure 4.2: An ID-Parameter for a Course PSE

```
GET Logic2/Professor/GradesStats?course_id=1
```

Figure 4.3: Another ID-Parameter for a Course PSE

Note that in both Figure 4.2 and Figure 4.3, the ID-Parameters have values that most likely represent objects in persistent state, but they do not represent the object that is likely being created in CreateQuiz or read in GradesStats (supposing this request were identifiably Read-classified). They each represent the "course" object identified by the ID 1 but this "course" object is not the item being targeted by the CRUD-action indicated by the two resource names. For example, the request in Figure 4.2 most likely performs a Create operation on a "quiz" object, as explained in Section 4.1.1. It is important to remember that requests may contain ID-Parameters for multiple Persistent State Elements, and that the PSE acted upon by the request (as determined by the resource name) may or may not be included in those Persistent State Elements provided via ID-Parameters. Additionally, many requests contained ID-Parameters representing ambiguous PSEs. I refer to these as *ambiguous ID-Parameters*. For example, the

`CreateProfessor` resource takes the parameters `username`, `email`, `fname`, and `lname`. The `username` parameter is an ID-Parameter but does not explicitly indicate the PSE it represents. In a majority of requests such as those for the `CreateProfessor` resource containing ambiguous ID-Parameters, the ambiguous ID-Parameter corresponded to an ID for the request PSE. In the `CreateProfessor` request, for example, the value of `username` is an ID string for a professor.

4.2.3.2 ORPM

The ORPM application contained the highest percentage of unique request combinations containing ID-Indicators. However, these ID-Parameters contained ID strings belonging to ambiguous PSEs. For example, Figure 4.4 displays one ambiguous ID-Parameter that appears in many requests for resources in the ORPM application.

```
POST ORPM/properties_view.php --post-data="&SelectedID=1&SelectedField=2&DisplayRecords=all"
```

Figure 4.4: Ambiguous ID-Parameter in ORPM

The `SelectedID` parameter in Figure 4.4 and similarly ambiguous ID-Parameter names are used in ORPM, making it difficult to predict the type of PSE the requests act upon without a personal knowledge of the application's implementation details. Although the ambiguity in naming parameters with respect to PSEs yields less predictive possibilities for the ORPM application than the Logic application, the high rate of ID-Indicators in its set of request combinations at least supports the concept that ID-Parameters can be identifiable by a set of relatively common ID-Indicators.

4.2.3.3 WebCollab

The WebCollab application revealed another important parameter naming possibility. Out of the 52 unique request combinations, 45 request combinations contained a parameter called `action`, which serves the same purpose in all 45 requests. The WebCollab request combinations contained requests for only 12 unique resources because

the application utilizes a dispatcher model. By this, I refer to a model where each resource possesses a switch statement with many cases, each of which deploys some different functionality based on the value of the parameter named `action`. Out of these 12 unique resources, only 4 resources do not require the `action` parameter. Those 4 are the resources used exclusively to perform user login and logout. Interestingly, the `action` parameter takes various verbs as values, according to usage data.

GET Webcollab/tasks.php?**action=show**&taskid=10

Figure 4.5: Action Parameter Example

Figure 4.5 displays the use of the `action` parameter. Like the resource name `tasks.php`, WebCollab features only resource names made up of a noun at a time. The `action` parameter is used to indicate which CRUD action a request will perform on the PSE indicated by the resource name. This pattern suggests that there are other ways to CRUD-classify requests than just looking at the resource name (as in the resource naming study). If a web application uses one consistent parameter to specify the type of action performed by its resource, I will refer to this parameter as an *action parameter*. To qualify as an *action parameter*, a parameter must appear in all of the requests whose responses have dynamically determined aspects, with the exception of the `login` page. It must also serve the same purpose in all the requests it appears in, like the `action` parameter in WebCollab.

4.3 Conclusions from Exploratory Studies

1. CRUD classifications for requests can be determined by either:
   (a) Checking the resource name for a PSIW
   (b) Identifying the value of the action parameter (if any such parameter exists for the application)
2. PSE objects of CRUD-classified requests, which I will refer to as request PSEs, can be determined by:

   (a) Finding the PSE name as part of the resource name (usually when the resource name also indicated the CRUD classification)
   (b) Finding the PSE as the *entire* resource name (only probable if the application uses an action parameter)
   (c) Finding the PSE name as a part of an ID-Parameter
3. IDs for request PSEs, or PSEs affected by a CRUD-request, are often passed as the values of ID-Parameters, that either:
   (a) Have a name containing an ID-Indicator when split
   (b) Have a name that is the name of a known PSE (usually just a noun)
   (c) Have a name that is a single ID-Parameter when split, and is the only such parameter (*ambiguous ID-Parameter*) in a request containing no clear ID-Parameter for the request PSE

Most importantly, the evidence gathered here suggests that a relatively universal system for naming resources and parameters in web applications exists. Although the three case studies evaluated three unique applications built on very different frameworks, all three applications seem to support the CRUD-classification of requests and identification of ID-Parameters in requests.

Chapter 5

APPROACH: CLASSIFICATION EXTRACTOR AND PSE-MATCHING ALGORITHM

Utilizing the PSE and CRUD-classification concepts discovered in the exploratory studies, I hypothesize that it is possible to create an algorithm that modifies an original input test suite to be more persistent-state aware. The use of CRUD classifications and PSEs will allow this algorithm to extract dependencies between a subject application's resources, dynamically map unique IDs to the PSEs they represent, and replace request parameters values with intelligent values to output a new persistent-state-aware test suite.

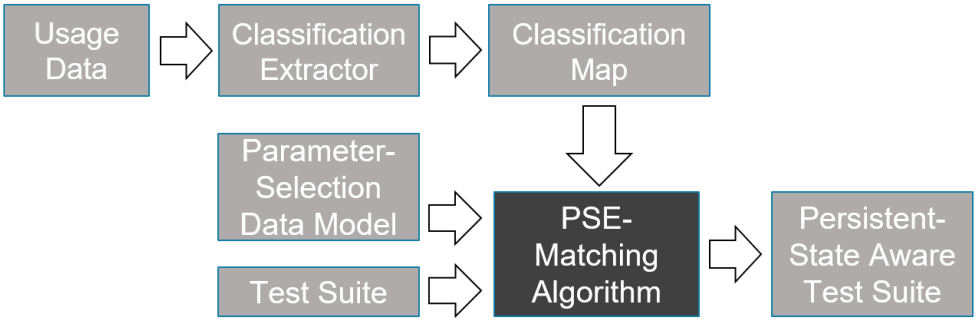

Figure 5.1: Input and output for the Classification Extractor and the PSE-Matching Algorithm

Figure 5.1 displays the inputs and outputs of the PSE-Matching Algorithm. The inputs for the algorithm, in greater detail, are:

- **A test suite** is a collection of files containing requests. The only request information needed to interact with the web application is the request type (Get, Post, Put, etc.), the request URL (without the query string in the case of a Get request), and the request parameters (both name and value). Each file within the test suite is one test case, made up of a series of such requests.

- **Usage data**, consisting of logged requests from real users, goes through preprocessing by the Classification Extractor before reaching the PSE-Matching Algorithm.
- **A parameter-selection data model** specifies the conditional probability of each parameter set from usage data appearing in each request. The data model can relate any number of conditions to the probability of encountering a certain parameter set. For example, a data model might use the parameters specified in the three most recent requests to determine the probability of the next request containing a set of parameters. Another data model might just use the resource name in the current request as its only condition for probability of the next request containing that set of parameters. The PSE-Matching Algorithm will use the specified parameter-selection model to acquire a parameter set for various requests as it traverses the input test suite.

The algorithm will traverse the test suite by iterating over its test cases in order, reading each request in search of PSEs that can be connected to unique ID strings. As the algorithm finds ID-to-PSE mappings, it will maintain a map of these ID-PSE pairs, which will represent the objects currently assumed to be in persistent state at any given point in the traversal of the test suite. When the algorithm comes across requests that try to read, update, or delete PSEs that do not exist (as determined by the mappings in the ID-PSE map), the algorithm uses the map of existing ID-PSE pairs along with the supplied parameter-selection data model to supply new parameter values for these requests.

The goal of the PSE-Matching Algorithm is ultimately to take *any* test suite as input, regardless of the algorithm used to create that test suite, and to return a more persistent-state aware version of that suite. In other words it aims to return a test suite with less logical inconsistencies than the original. It follows that a more persistent-state aware test suite should achieve higher line and branch coverage of subject application code. To achieve greater coverage results, it is important that the algorithm infers connections between parameters and resource names without making parameter value replacements that inadvertently detriment the coherence of the test suite. It is for this reason that the algorithm takes any data model as input to guide and/or restrict it in selecting parameter values as it traverses the test suite.

Table 5.1 provides a summary of the terminology that will be used frequently throughout the rest of the description of the PSE-Matching Algorithm.

5.1 Classification Extractor

Before the PSE-Matching Algorithm can modify an input test suite, another algorithm first takes usage data, including logs of requests from users, as input and compiles the unique requests from the logs. From these requests, the classification extractor creates a map that maps request characteristics to the CRUD classification and request PSE for requests containing those characteristics. For different application types, the information necessary to extract a CRUD classification and PSE differs, therefore the actual classification map created by the extractor will differ slightly. However, in both cases the *classification map* is a mapping that uses the fewest possible components of a request to predict request PSEs and classifications. The use of this map will make the PSE-Matching Algorithm much more efficient by saving it the work of re-classifying requests as it traverses a test suite. Instead of classifying and extracting a PSE from each request in a large test suite, the PSE-Matching Algorithm can reference the map provided by the classification extractor to provide it the request classification and request PSE. For applications like WebCollab whose requests rely on an *action parameter* to specify the parts of the resource's code to exercise, the classification extractor determines a classification and PSE for each request based on the request's resource name, parameter names, and action parameter value (if an action parameter exists in request). If a request's action parameter value is a PSIW or contains only one PSIW when split, the classification extractor determines that the request itself has the same CRUD classification as that PSIW. If the resource name can be split to only contain one word, the classification extractor determines that this word is the PSE of the request. As a result, the classification extractor adds an entity to its output map whose key is the resource name, parameter names, and action parameter (if applicable). This key maps to a string containing the classification and PSE for any request containing the specified resource, parameters, action parameter value.

Persistent State Indicator Word (PSIW)	• "action word" indicating CRUD action performed • has consistent CRUD classification across web applications • request whose resource name contains a PSIW is given that PSIW's CRUD classification
Persistent State Element (PSE)	• object predicted to be in persistent state by some part of a request • parameters whose names can be split into a noun and ID word or just a standalone noun are said to have that noun as their PSE • see *request PSE* for the case when PSE found in resource name
Request PSE	• direct object of CRUD-action performed by CRUD-classified request • always determined by finding noun in resource name
ID-Indicator	• Word that, when found within a parameter name, indicates corresponding parameter value is unique ID string
ID-Parameter	• Parameter whose value is likely to be the unique ID key for a PSE • Parameter name is either a known PSE or contains an ID-Indicator along with a known PSE
Persistent State Map	• Map connecting PSEs to IDs belonging to that PSE
Action Parameter	• Parameter appearing in a majority of an application's possible requests, whose value specifies the action performed by each request

Table 5.1: Terminology for the Classification Extractor and the PSE-Matching Algorithm

For applications like Logic whose resources are single-responsibility and do not make use of action parameters, the classification extractor determines a classification and request PSE for each request based on the resource name alone. It does so by checking for a PSIW in the resource name when split. If the resource name is made up of a PSIW and one other word, the classification extractor determines the PSIW's classification to be the request's classification and the other word to be the request's PSE. The classification extractor for this type of application maps each resource name to a string containing a classification and PSE for any request for this resource name.

For both application types, any request information that does not conclusively result in the extraction of either a classification or a PSE will simply map to an empty string because the PSE-Algorithm will be unable to predict the persistent state implications of a CRUD action on an unknown PSE or of an unknown CRUD action on a PSE.

Figure 5.2 shows an example of how the classification extractor adds items to the classification map given a request within a user session. Note that Figure 5.2 illustrates this process for Logic, an application with single-responsibility resources and no *action parameters*.

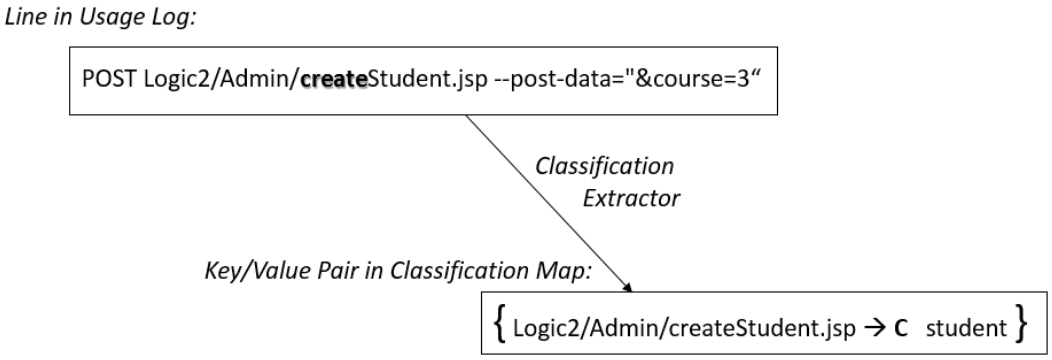

Figure 5.2: Classification Extractor example on Logic request

5.2 PSE-Matching Algorithm

After the Classification Extractor has turned usage data into request classification data, the classified requests and their PSEs are passed to the PSE-Matching Algorithm along with an input test suite to be made more persistent-state aware. The primary state maintained by the PSE-Matching Algorithm is a map of IDs to persistent state elements which I call the *Persistent State Map*. The *Persistent State Map* maintains the set of ID/object pairs in existence throughout the traversal of a test suite. Each key in the Persistent State Map is a string made up of an ID and the object, or PSE, it represents. An example key is "74 quiz" for a quiz with ID 74. Its corresponding value is an object containing various predicted attributes of the PSE represented by the ID. For example, if the most recent action performed on the quiz with ID 74 is an update, this data will be stored in the state of the object with key "74 quiz" in the Persistent State Map.

The PSE-Matching Algorithm consists of many rules, each of which can modify the Persistent State Map or modify the test suite itself based on its findings within the Persistent State Map as it traverses the test suite. If, during the test suite traversal, the algorithm encounters a parameter whose value should be changed, it attempts to use the Persistent State Map to produce a list of more likely parameter values. If the algorithm succeeds in producing a list, I will refer to these more likely parameter values as *eligible parameter values*.

When the algorithm selects eligible parameter values for a certain request parameter, the algorithm does not necessarily make a replacement. At the outset of the algorithm, the algorithm takes as input a "randomness threshold" between 0 and 1 that tells the algorithm how often (by percentage) to leave an original value in the test suite when it comes up with more eligible values. Randomness ensures that if one logical inconsistency is found many times throughout the test suite, the algorithm does not necessarily remove the inconsistency entirely. By maintaining the inconsistency sometimes, the algorithm maintains some coverage of the error code executed by that inconsistency, since it is important to test error code too. Furthermore, the algorithm

may produce a large number of eligible parameter values for parameter x in request r. The PSE-Matching Algorithm utilizes some user-specified parameter selection model to choose a parameter value set from usage data for request r that contains one of the eligible parameter values as the value of parameter x. Therefore, even though a parameter value is deemed eligible, the algorithm does not necessarily insert the parameter value into the test suite in place of the original parameter value. Eligible values found within more statistically common parameter sets for request x may be more likely to be selected by the PSE-Matching Algorithm if the specified parameter-selection model is reflective of selection frequencies from usage data. Then, if a parameter value set containing an eligible value is selected, the entire parameter set is placed into the request as parameter values. If the parameter value set supplied does not contain an eligible value for x, a new parameter value set is chosen–either until the selected set contains an eligible value for x or until all possibilities are exhausted. If all possibilities are exhausted and the model is unable to come up with a parameter value set with an eligible value as the value for parameter x, the PSE-Matching Algorithm will leave request r unchanged. When the PSE-Matching Algorithm refers to the parameter-selection model to acquire a parameter value set containing an eligible value for x, I will refer to the final parameter value set chosen as the *eligible parameter set* if the algorithm succeeds in acquiring one.

5.2.1 Example of PSE-Matching Algorithm

The following example illustrates the identification of key components of a request, the algorithm's interpretation of those components, and the changes the algorithm makes to the original request as a result.

Figure 5.3 shows the components of a request, as identified by both the classification extractor and the PSE-Matching Algorithm. Before the PSE-Matching Algorithm even comes across this request in the test suite, the classification extractor has already determined that any request for `tasks.php` with the parameters `action` and `taskid` as well as the action parameter value "delete" will be a D-classified request with "task" as

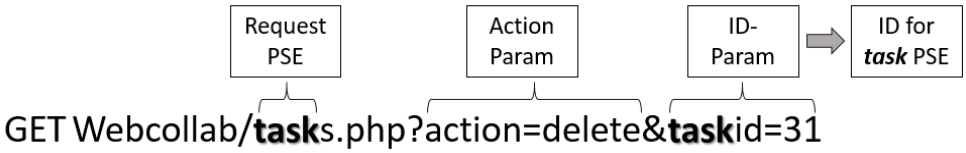

Figure 5.3: Identifying key elements of a request

its PSE. Thus, the PSE-Matching Algorithm finds, by checking the classification map, that this request acts on the PSE "task" and is Delete-classified. The PSE-Matching Algorithm then identifies `taskid` as an ID-Parameter because its name is parsed into the words "task" and "ID", which is an ID-Indicator.

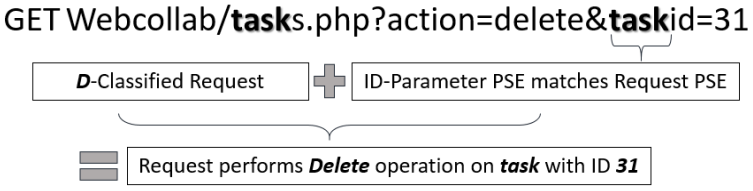

Figure 5.4: Inferring the request's effect on entities in persistent state

In Figure 5.4, the PSE-Matching Algorithm infers information about the *specific* object this request deletes. To successfully delete an object from an external data store, that object must first exist in the external data store. This means that if it is possible to identify the precise object deleted by the request, the algorithm must make sure that particular object exists in persistent state. Since there is an ID-Parameter (`taskid`) in the request that has the same PSE as the overall request itself, the specific PSE specified by the `taskid` parameter is predicted to be the object deleted by the request. Specifically, the request is predicted to perform a delete operation on a task with ID 31.

The next step is to check the PS-Map to see if a task with ID 31 exists in persistent state. A task with ID 31 does not exist in persistent state, and as a result the PSE-Matching Algorithm will check the Persistent-State Map to find any task

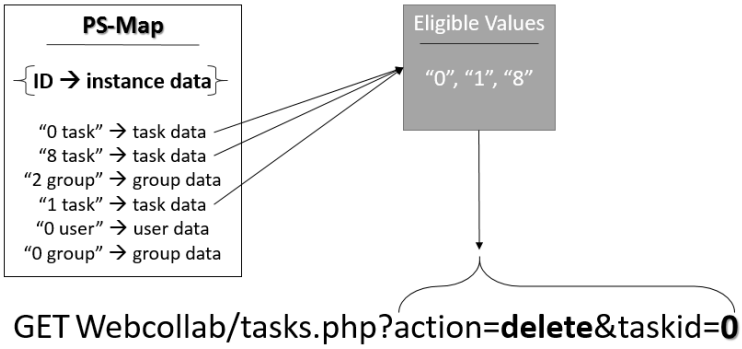

Figure 5.5: Selecting eligible parameter values for the request

values on which the request can act. Figure 5.5 depicts the process. The algorithm compiles a list of eligible values containing the current IDs for tasks in the map. These 3 eligible values are added to the list, and one final output value is selected to become the new `taskid` value, based on probabilities indicated by the parameter-selection data model input to the PSE-Matching Algorithm. In Figure 5.5, the new ID-Parameter value is 0.

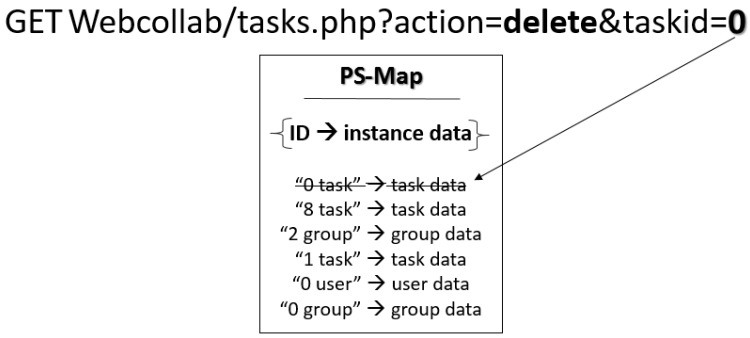

Figure 5.6: Updating the Persistent-State Map upon acquiring an eligible parameter set

Finally, the deleted object must be removed from the Persistent-State Map, as shown in Figure 5.6, because the request is Delete-Classified and this ID-Parameter `taskid` indicated the task targeted by this delete operation. This example illustrates

the primary function of the algorithm– to track major changes to persistent state and direct the test suite to meet persistent state requirements with the resulting insights.

5.2.2 PSE-Matching Algorithm Rules

The following more fully details the changes the PS-Matching Algorithm makes to both the input test suite and the Persistent-State Map as it traverses the input suite.

Create-Classified Requests:

Since Create requests often require a unique ID-Parameter value to represent the new entity created then added into persistent state, it is important that a test suite succeeds at some point in providing unique IDs as ID-Parameter values.

POST Logic2/Professor/CreateQuiz --post-data="&course=1&quiz_id=192&quiz_title=Graded+Quiz"

Figure 5.7: Create-classified request creating an entity already in persistent state

If a request is classified as a Create request, the request is checked for a PSE, which will be the object on which the entire request acts. In the example of Figure 14, "task" was determined to be the request PSE. In Figure 5.7, "quiz" is determined by the Classification Extractor to be the request PSE, or the object on which the request likely acts. Since it matches the PSE of ID-Parameter `quiz_id`, the PSE-Matching Algorithm infers that the request is performing a create operation on the *specific* quiz object with ID 192.

Since this request is Create-classified, the algorithm must check to make sure there is not already a quiz object with ID 192 in the application's persistent state. It does so by consulting the Persistent-State Map for the key "192 quiz". If no such item is found in the map, the algorithm makes no change to this request. Otherwise, if "192 quiz" is found, the algorithm must create a list of eligible values for the parameter `quiz_id`. For Create-classified requests, this means the algorithm gathers all logged sets of parameters for this POST request for the CreateQuiz resource (which also have

the parameters `course`, `quiz_id`, and `quiz_title`). Of these possible parameter sets, it gathers as many as possible whose `quiz_id` parameter value is not associated with a quiz currently in the PS-Map. These `quiz_id` values become the eligible values for this parameter. From the list of eligible values, a new quiz ID value is selected along with the rest of its eligible parameter value set using the parameter-selection model supplied.

If a new eligible parameter value set is successfully chosen to replace the original parameter set, the algorithm finally adds the new value of the ID-Parameter to the Persistent State Map to indicate that this ID has now been used to identify the request's PSE. After replacing "192" with some ID x in the Figure 5.7 request, the algorithm will add a quiz with ID x to the PS-Map.

In any case where the request PSE does not match the PSE of an ID-Parameter in the request, the algorithm makes no changes. This is because even if there are other ID-Parameters in the request, a Create-classified request may take new or old IDs for objects other than the one on which it performs a create operation. Due to this ambiguity, the only time a Create-classified request changes the parameter set is in request like Figure 18 where the ID of the object on which the request acts is unambiguously identifiable.

Read-Classified and Update-Classified Requests:

Since Update and Read requests act on objects which already exist in persistent state or elsewhere, it is important that valid ID values of objects in persistent state are passed to ID-Parameters in order that they access objects in existence, thus likely accessing non-error code to access or update objects.

GET Webcollab/**task**s.php?action=show&**taskid**=35

Figure 5.8: Read-classified request accessing nonexistent entity in persistent state

In a request like that in Figure 5.8, where the request PSE matches the PSE of an ID-Parameter in the request, the ID for the PSE operated on by the request is known. In Figure 5.8, the Read-classified request has PSE "task" and the ID-Parameter `taskid` has the same PSE. Therefore, the value of this ID-Parameter is predicted to be the ID of the object on which the read operation is performed. Here, the request performs a read operation on the task object with ID 35. Note that if the request were Update-classified, the same process would apply.

Next, the PSE-Matching Algorithm checks for an object with the specified ID in the PS-Map. In the case of the Figure 5.8 request, the algorithm checks the PS-Map for a task object with ID 35. If it exists in the map, the value for `taskid` will remain unchanged because it already successfully reads existing data from persistent state. However, if a task object with ID 35 is not located in the map, the algorithm must compile a list of eligible parameter values for the `taskid` parameter. If it successfully finds IDs belonging to task objects in persistent state, these IDs make up the list of eligible values. From these, a new ID value along with the eligible parameter value set containing it is chosen.

In a case unlike Figure 5.8, where perhaps none of the ID-Parameters found in the request obviously have the same PSE as the request, the algorithm first evaluates whether an ambiguous ID-Parameter exists that might represent the request PSE. If not, the algorithm tries to replace the parameter set with one that supplies existing IDs for any other ID-Parameters in the request. For example, Figure 5.9 depicts a situation in which multiple ID-Parameters are identified but none obviously has the same PSE as the request itself, whose PSE is "student".

Figure 5.9: Update-classified request in which no ID-Parameter for the Request PSE exists

In Figure 5.9, since neither of the two ID-Parameters, `course` or `username`, can unambiguously be predicted to represent the ID of the "student" PSE, which is

the PSE of the request itself, the parameter set is further evaluated. The `username` parameter is ambiguous because it does not clearly reference a PSE. However, since it is the only ambiguous ID-Parameter in the parameter set and the parameter set does not contain any clear ID-Parameter for "student", the PSE-Matching Algorithm can infer that the `username` ID-Parameter takes the username of a "student" object. Thus, if a student with username "bs1" does not exist in the PS-Map, the algorithm searches for eligible values for the `username` parameter and makes a replacement the same way it did for the `taskid` parameter in the Figure 5.8 example.

However, in the case that the `username` parameter did not exist in the Figure 5.9 request, the PSE-Matching Algorithm would not be able to select any parameter in the set to represent the request PSE, "student". In this case, it would attempt to ensure that it provides an existing course ID for the value of the `course` parameter, if any course IDs exist in the PS-Map. In this case, the algorithm checks if a course object with ID 4 exists in the PS-Map. If so, the algorithm makes no changes because the request already refers an existing entity. If no course object with ID 4 is found, the algorithm compiles a list of eligible parameter values for the `course` parameter. The list will simply consist of IDs for "course" objects located in the Persistent-State Map. From the eligible values list, the parameter interactions model is used to select a final parameter value for `course` along with an eligible parameter value set to be placed into the request. Even though "course" is not the PSE this request is updating, it is unlikely that an Update or Read classified request would try to supply a new ID instead of providing the ID for an existing object regardless of whether the object is directly changed or read.

Note that neither Figure 5.8 nor Figure 5.9's request results in any change to the PS-Map. This is because update and read operations are likely to change objects already in application state and unlikely to result in those objects removal. The only case that will cause the PS-Map to become faulty is an Update-classified request that changes a PSE's unique ID string. Although this functionality would ideally be met with a change in the PS-Map, the PSE-Matching algorithm has no handling for such

requests as this ID-modifying functionality is difficult to predict using language alone.

Delete-Classified Requests:

Because a Delete-classified request also seeks to perform an action on an existing object in the same was a Read or Update request does, there is only one difference between the way the algorithm treats Delete requests and Read or Update requests. The one important difference is the modification of the Persistent-State Map. No matter which particular parameter value is replaced in Read and Update requests, the Persistent-State Map remains unchanged because no object is created or removed. However, Delete requests remove data from persistent state and should therefore affect the Persistent-State Map by deleting the IDs of the deleted object(s). As portrayed in Figures 5.3- 5.6, when a request contains a PSE as well as an ID-Parameter with that same PSE, it is possible to predict which object the request removes from the shared data store. In Figure 5.6, the task object with ID 0 is removed because the PSE of the Delete-classified request is "task" and the ID-Parameter with the same PSE points the algorithm to the specific ID of the task being deleted. This allows the algorithm to make that same change to its own model of the application's state, the Persistent-State Map.

In cases like the Figures 5.3- 5.6 example, in which the algorithm can infer the actual ID of the deleted object, the only replacement the PSE-Matching Algorithm will ever make is a replacement like the one in Figure 5.5 that targets the ID-Parameter of the deleted object. In other cases where an ID-Parameter is found in the request but it does not necessarily represent the deleted item, the algorithm changes the parameter value so it contains an ID currently in existence for the parameter's PSE, if possible.

GET Webcollab/**forum**.php?action=delete&**postid**=10

Figure 5.10: Delete-classified request in which no ID-Parameter for the Request PSE exists

Figure 5.10 shows a situation in which a request results in no change to PS-Map. This is because the ID passed by this request is an ID for a "post" object and the thing being deleted is predicted to be a "forum" object (although this is an incorrect prediction, it does not cause any harm since nothing is removed from the map). Since no ID for a "forum" PSE appears in the request, the PSE-Matching Algorithm checks the PS-Map for a post object with ID 10. If found, the algorithm makes no replacements. Otherwise, the PSE-Matching Algorithm attempts to build a list of eligible values for `postid` and makes a parameter-set replacement using the parameter-selection model if possible.

Chapter 6

EXPERIMENT, RESULTS, AND ANALYSIS

In this chapter, I present the research questions I seek to answer experimentally. I then detail the experiment conducted to test these research questions and describe the metrics by which I measure the PSE-Matching Algorithm's success. I then provide the results attained and an analysis of those results, as well as an analysis of the time complexity of the algorithm. I then describe the threats to the validity of the experiment's results. I close with my observations and conclusions.

6.1 Research Questions

I seek to answer the following questions in this chapter:

1. Does the PSE-Matching Algorithm modify an original test suite to make it persistent-state aware in a way that results in increased code coverage as compared to the original test suite?
2. Can analysis of the coverage improvement attained by the PSE-Matching Algorithm reveal particular strengths or failures of the algorithm?
3. What does the PSE-Matching Algorithm cost with respect to time and space?

6.2 Experiment Methodology

I designed an experiment to answer the above research questions.

To test the effectiveness of the PSE-Matching Algorithm, I created test suites of varying size for both Logic and WebCollab as subject applications (described in Chapter 4) and evaluate the improvement achieved by passing each suite as input to the PSE-Matching Algorithm.

For each experiment, I first create a test suite using the Sant et al 2-gram simple data model [9], which builds test cases one request at a time using a 2-gram Markov

chain to determine the next request based on the one previous request. I describe this data model in greater detail in Subsection 6.2.1. For each application, I generate a test suite containing 50 cases, one containing 100, and one containing 150.

I then instrument the code of the subject application and replay this original test suite, using a code coverage tool to track and report the coverage achieved by the suite. For WebCollab, which is written in PHP, I use a modified version of PHPCoverage [2], and for Logic, which is written in Java, I use Cobertura [3]. Since PHPCoverage does not report branch coverage, I will only be reporting line coverage for WebCollab instead of both line and branch coverage, as reported for Logic. The coverage attained by this first test suite will be the coverage value to which I will compare the coverage achieved by a persistent-state aware suite.

Next, I will pass this test suite into the PSE-Matching Algorithm along with usage data for the subject application. I will pass the Parameter-Interaction Model to the PSE-Matching Algorithm as the parameter-selection data model. The Parameter-Interaction Model is described in Subsection 6.2.2. Then, I replay the new persistent-state aware test suite using a coverage tool to track and report the coverage achieved by the suite. I hypothesize that comparison of the coverage achieved by the input test suite and the output test suite should reflect an improvement in coverage of resources and code responsible for interacting with persistent state.

6.2.1 N-Gram Simple Model

The goal of usage-based testing is to leverage real user sessions to test realistic usage patterns as well as to extrapolate additional realistic patterns for testing not explicitly found in those sessions. To do this, it is necessary to employ a data model that is informed by–but not confined to–patterns in usage data. The use of Markov assumptions in building test suites achieves this balance well. A Markov assumption refers to the probability of one event occurring based on the previous n-1 events. Given a certain value for n, a data model that determines the probability based on the preceding n-1 events is called an n-gram model. For the sake of building test suites, we

are interested in the probability of a certain request occurring based on the requests preceding it.

The n-gram simple approach to test suite generation, proposed by Sant et al [9], builds a template test suite one request at a time using an n-gram data model. Recall that a template test suite is a suite whose requests do not have parameter values. A unigram (1-gram) model, for example, chooses the next request in a case based only on the probability of each request happening independently of previous requests, while the bigram (2-gram) model chooses the next request as a function of the one request preceding it. Although a unigram (1-gram) model achieves high code coverage the most rapidly in comparison to other n-gram models, it creates the least realistic test cases in terms of application functionality successfully exercised. On the other hand, the bigram and trigram models both succeed in accessing application code that modifies state, as well as application code responsible for exception handling [9]. Upon completion of the template suite, the Simple Data Model is used to place parameter values into the template requests within the suite. The Simple Data Model utilizes a 2-gram Markov assumption model to select the parameter set supplied in each request. It determines the probability of each parameter set based on the current request's resource and parameter names.

Using a test suite generated with the 2-gram and simple data models as input to the PSE-Matching Algorithm is consistent with the goal of exercising realistic usage patterns and functionalities within the subject application, including functionalities that interact with persistent state. A test suite generated with these models will successfully access persistent state occasionally because the navigation path through the application is realistic and the parameters selected for requests pseudo-randomly are sometimes valid. Such a test suite lends itself well to improvement by an algorithm focused on optimizing parameter selection.

6.2.2 Parameter Interaction Model

Previous research in web application testing has offered various approaches to providing test suite templates, or test suites whose requests do not yet contain parameter values, with concrete parameter values. The *parameter interaction* data model, offered by Sant et al [9], utilizes probability determined from usage data to choose the *set of* parameter values for each request in a suite.

The model associates each resource name in usage data with the collection of parameters passed in requests with it. Each time a request for a certain resource appears in the usage logs, each unique set of parameter values supplied in the request becomes a parameter value set for that resource. Once these parameter value sets are compiled for each resource, each resource r will be associated with x parameter value sets. Of these x parameter value sets, those that appear in many requests for resource r will have higher usage probability than those that appear less.

Using the *parameter interaction* model to choose parameter values for requests instead of an independent model that supplies parameter values individually (i.e. not in a set) allows the benefit that the resulting requests model actual requests made by users. Leveraging actual, recorded parameter sets reduces the likelihood that logical inconsistencies will exist between parameters within a request. For example, in a login request that requires a username and a password, the parameter interaction model will supply parameter values more likely to enable successful login by keeping passwords and usernames together.

Resource and Parameter Names	Sets of Parameter Values	Probability
ORPM/properties.php –ownerID – property	user100, 23419 admin, 19294	50% 50%
ORPM/editProperty –property –owner –ptype	property1, admin, condo property2, user100, apartment property2, user193, condo	25% 50% 25%

Figure 6.1: Example of conditional probabilities for selecting parameter sets

Figure 6.1 shows an example of the conditional probabilities used to select parameter value sets with the Parameter-Interaction data model. It displays the likelihood of selecting all possible parameter sets for two ORPM requests. The probability of selecting user100 as `ownerID` and 23419 as `property` for a request containing the `properties.php` resource is 50% because this reflects the trends found in usage data. The second parameter set for the `editProperty` resource appeared twice as often as the other parameter sets for this resource in usage data, and therefore has twice as high a likelihood of becoming the parameter set used in a test suite.

6.3 Metrics

The first metric by which I evaluate the PSE-Matching Algorithm is coverage improvement. First I measure the coverage achieved by an original input test suite and then measure the coverage achieved by the same suite after it is modified by the PSE-Matching Algorithm. I will refer to the original as the Simple test suite and the modified version as the *Persistent State Aware (PS-Aware)* test suite. I will use both line and branch coverage in evaluating the algorithm's success on the Logic application and only line coverage for the WebCollab application due to limitations imposed by PHP coverage tools. *Line coverage* is the number of executable lines of application code exercised by the test suite. *Branch coverage* is the number of possible branches executed, where a branch refers to the specific set of conditions met in order for the application code to enter each control structure. For example, consider some application code containing an *if-else* statement and a test suite that manages to execute the *if-else* statement only once, where the value of its boolean condition evaluates to True. Then for this particular code segment, the suite has achieved 50% branch coverage because the boolean condition has never evaluated to False.

As part of my analysis of this coverage improvement data, I provide the results of a coverage evaluation script designed to identify potential oversights and strengths of the algorithm. The evaluation considers differences between the original input test suite and the algorithm's output suite. Given a resource X, the line coverage for that

resource achieved by the input test suite, *LCOV(I(X))*, and the line coverage for that same resource achieved by the output Persistent-State Aware test suite, *LCOV(O(X))*, I define X's ratio of improvement Δ as a decimal between -1 and 1 (inclusive) that indicates proportionally how much the algorithm improved coverage of resource X. The function for determining Δ of resource X is shown in Figure 6.2.

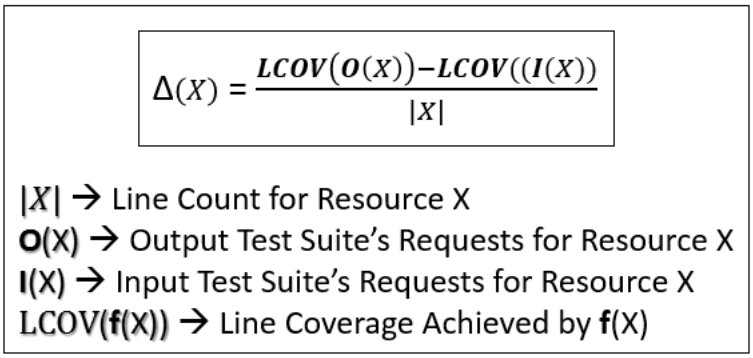

Figure 6.2: Function for coverage improvement threshold

Values of Δ less than 0 indicate negative line coverage improvement accomplished by the algorithm, which is undesirable. However, depending on the expectations of testers, a certain value for Δ might be very desirable for one tester or subject application and very undesirable for another tester or application. Therefore, the remaining heuristics allow the tester to identify a value L such that any resource X achieving $\Delta(X) < L$ is considered a low-improvement resource under the algorithm that took *I(X)* to *O(X)*.

Given some input between -1 and 1 for L, the evaluation script begins by compiling all the names of low-improvement resources, or any resource X achieving $\Delta(X) < L$ by Δ defined in Figure 6.2, as well as high-improvement resources, or any resource X achieving $\Delta(X) > L$ by the same definition of Δ. It then splits the names of all the low-improvement and high-improvement resources into words (using splitting algorithm mentioned in Exploratory Studies), and checks for possible PSEs that appear more than once in the group of resources. Instead of consulting the Persistent-State

Map to get names of known PSEs of low-improvement resources, the script simply splits resource names into words and outputs any component word that is in the English dictionary but not in the list of PSIWs. This allows for the identification of PSEs the algorithm may not have recognized as PSEs automatically that should have been PSEs.

After providing this analysis of the coverage data, I will discuss the time and space requirements of the Classification Extractor and PSE-Matching Algorithm. I will present the time required to turn usage data into classifications and PSEs by the classification extractor as well as that required to apply these classifications to input test suites of various sizes by the PSE-Matching Algorithm. I will provide a time complexity evaluation of both algorithms as well.

6.4 Results and Analysis

In this section I provide the data acquired about the PSE-Matching Algorithm using the metrics discussed in Section 6.3. Upon passing each suite into the PSE-Matching Algorithm and acquiring the coverage improvement over the original, I record the coverage achieved by both the original and the PS-Aware test suite. For Logic, I report the average coverage achieved before and after modification by the PSE-Matching Algorithm across 10 suites at each size (50, 100, and 150 test cases). For WebCollab, I report the coverage achieved by one test suite at each size before and after modification by the PSE-Matching Algorithm.

6.4.1 Improvements in Line and Branch Coverage

In Figures 6.3 and 6.4, I show the average coverage improvement made by the PSE-Matching Algorithm on ten test suites of each size for Logic. Each test suite modeled a series of sessions of users with administrative privileges because these users perform the most CRUD actions on data in persistent state. The coverage rates reported are a measurement of the lines/branches executed out of the total lines/branches in the admin JSPs and servlets, containing 5,245 lines and 1,950 branches total.

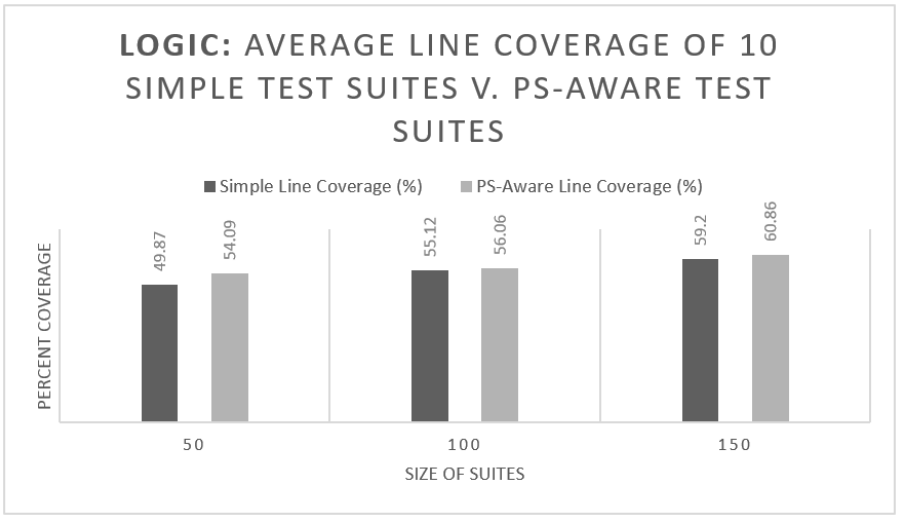

Figure 6.3: Average line coverage of 10 simple test suites v. PS-aware test suites with Logic as subject application

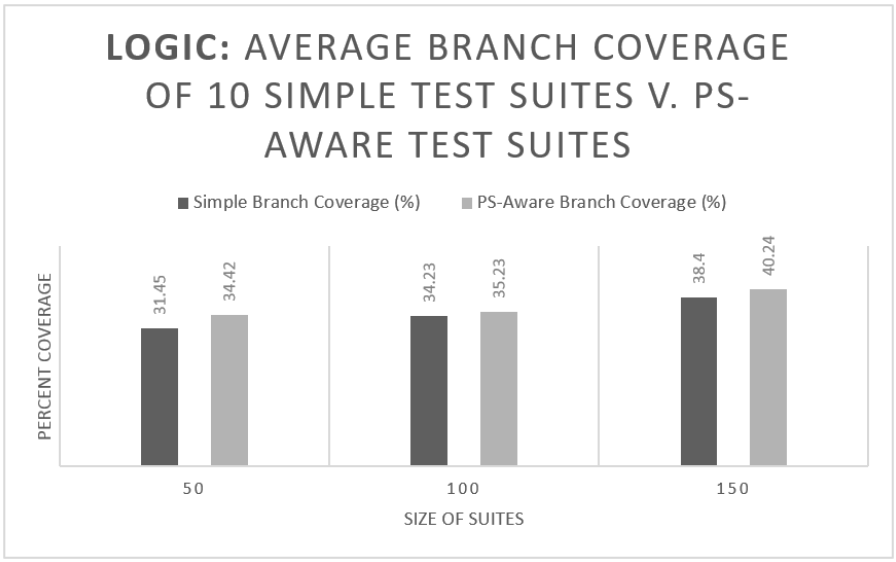

Figure 6.4: Average branch coverage of 10 simple test suites v. PS-aware test suites with Logic as subject application

The average improvement in line coverage achieved by making a 50-case test suite persistent state aware is 4.29%, a 100-case suite 0.94%, and a 150-case suite 1.64%. The lack of clear trend present in these improvements as size increases may be attributable to the small experiment size; however, the notably high coverage improvement achieved in making the average 50-case suite persistent state aware is worth evaluating.

The 4.29% increase in line and 2.97% increase in branch coverage for the average 50-case test suite reflects the influence of one test suite whose improvement by the PSE-Matching Algorithm was a large outlier. One single 50-case test suite saw its line and branch coverage more than double when improved by the PSE-Matching Algorithm. No improvement of any 100-case test suite or 150-case test suite was nearly as substantial as this. Figures 6.5 and 6.6 show the best experimental case for coverage improvement achieved on a test suite of each size, out of the ten in each set.

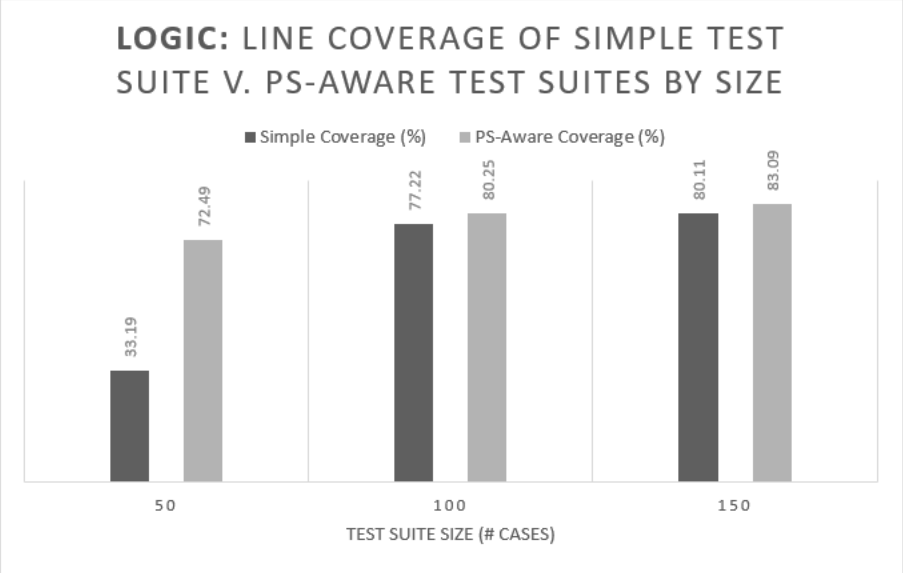

Figure 6.5: Line coverage of most improved Simple test suite v. PS-Aware test suite by size with Logic as subject application

Figure 6.7 shows the line coverage achieved by one Simple test suite of each size (50, 100, and 150) before and after modification by the PSE-Matching Algorithm

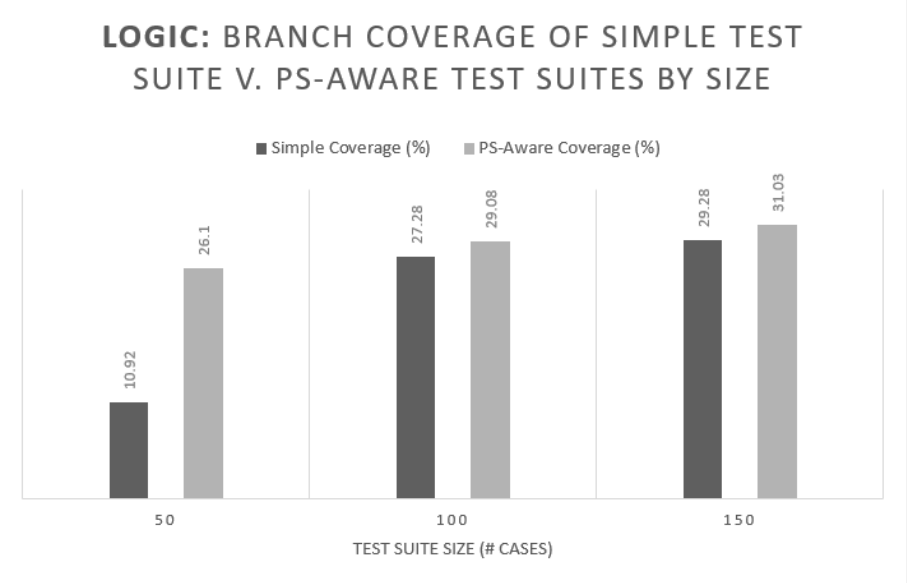

Figure 6.6: Branch coverage of most improved Simple test suite v. PS-Aware test suite by size with Logic as subject application

with WebCollab as a subject application. Because the version of PHPCoverage used in instrumenting WebCollab and tracking coverage caused the web application to respond excessively slowly, it was only possible to attain coverage data for one pair of test suites at each size.

6.4.2 Analysis of Coverage Data

While the coverage improvement achieved by the PSE-Matching Algorithm for both subject applications is not large on average, the results show that the smallest test suite has the largest potential for improvement. This trend requires further analysis. Table 6.1 represents the number of requests for each resource within the average WebCollab test suite of size 50.

The resources listed in Table 6.1 are the only PHP files in the WebCollab application accessible directly by request. Other PHP code in the application is accessed when files are included in the source code of these 13 resources. Therefore, although there are 100 PHP files containing executable code in the application, 87 of those files are accessed when certain conditions are met by the parameters passed into requests for

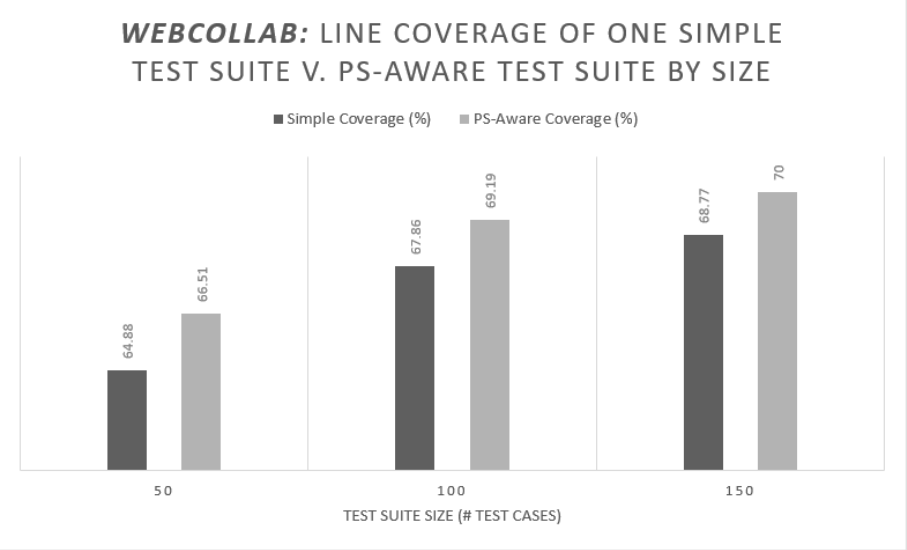

Figure 6.7: Line Coverage of Simple test suites v. PS-Aware test suites by size with WebCollab as a subject application

Resource Name	Av. Requests
Index.php	109
Main.php	95
Tasks.php	322
Users.php	85
Archive.php	5
Logout.php	38
Usergroups.php	39
Admin.php	8
Taskgroups.php	119
Contacts.php	9
Forum.php	24
Calendar.php	8
Files.php	5

Table 6.1: Number of requests per resource in 50-case WebCollab test suite

those in Table 6.1. Therefore, the only way for a test suite to access as many of these behind-the-scenes files in as many meaningful ways as possible is to vary its selection of request parameters.

It follows, then, that as suite size increases, the pseudo-random parameter selection method employed by the simple test suite generator eventually succeeds in selecting parameters that access many of these resources by brute force. The pseudo-random parameter selection may account in part for the higher rate of coverage improvement achieved by the PSE-Matching Algorithm for the 50-case suite in comparison with the 100 and 150 -case suites. Table 6.1 displays the large disparity the Simple algorithm creates between statistically frequently accessed resources and rarely accessed resources in usage data. The `Files.php` resource, for example, appears only 3 times in the usage logs in comparison with the `tasks.php` resource, which is accessed at least 3 times per session in the usage logs. The frequency with which each resource appears in the test suite is therefore a relatively accurate extrapolation of the usage data. However, a suite containing only 50 test cases gives the Simple algorithm very few opportunities to stochastically supply valid parameters to requests for lesser-accessed resources within the test suite. This creates a need for deliberate assignment of parameter values in generating a test suite that successfully interacts with external data. This need for deliberate parameter value selection especially in smaller suites explains why the smallest test suite was most improved by the PSE-Matching algorithm for WebCollab, as well as why one of the smallest test suites for Logic was the outlier for exceptional improvement by the PSE-Matching Algorithm. Although the improvement for the 50-case WebCollab test suite is small percentagewise, the 1.63% improvement represents an improvement of 326 lines of executable code across 27 resources.

This higher comparative rate of success achieved on smaller test suites for both applications illuminates a potentially useful characteristic of the PSE-Matching Algorithm: given a test suite that does not excessively exercise some particular application functionality, the PSE-Matching Algorithm seeks to make the limited accesses of that functionality as coherent as possible and therefore improve test suite efficiency. This

may have promising implications for testing much larger applications whose entire functionality cannot be as easily exercised by a small test suite. In such an application, it is more unrealistic to achieve high code coverage of certain resources by brute force and is therefore more critical to ensure that requests are predominantly successful in exercising non-error code.

6.4.3 Analysis of High and Low -Improvement Resources

Evaluation of the high- and low-improvement resources, as defined by the improvement threshold shown in Figure 6.2, with the threshold for improvement being 0.05 (a 5% increase or decrease in coverage), illuminates strengths and shortcomings of the PSE-Matching Algorithm. To gather this data, I gathered a list of the resources that were most commonly high-improvement resources and those that were most commonly low-improvement resources for each of the three groups of test suites (grouped by size of suite). For each, I provide the two PSEs appearing most frequently in the high-improvement resources. I then provide the two classifications most frequently belonging to high-improvement resources. I then provide the most frequent suggestion for potentially missed PSEs for each size. Table 6.2 displays this data.

Suite Size	**PSEs of Most Frequently Improved Resources**	**Classification of Most Frequently Improved Resources**	**Most Commonly Suggested PSEs**
50	Professor Quiz	Update	*None*
100	Professor Student	Update	*None*
150	Professor Course	Update	*None*

Table 6.2: Significant data from high-improvement and low-improvement resources from Logic's 30 test suites

The fact that all three groups of test suites had their most frequently improved resources contain the same PSE indicates that there is a degree of consistency in the modifications made by the PSE-Matching Algorithm. This is also indicated by the fact that the most frequently improved resources for all three groups possess the same classification. Update-classified resources were consistently the most improved, regardless of suite size. Surprisingly, there were no high-improvement resources in any of the 30 total test suites with a Read classification. Additionally, only the 150-size test suites possessed a Create-classified high-improvement resource. This indicates that the most consistently improved functionality by the PSE-Matching Algorithm is functionality that updates or deletes objects in persistent state. Despite the PSE-Matching Algorithm leaving many resources without coverage improvement, the evaluation of coverage data yielded no predictions for possible PSEs missed by the algorithm. This means that the lack of coverage improvement achieved was most likely not a result of errors in extraction of request PSEs. In Section 6.6, I examine possible causes of the lack of coverage improvement.

Since WebCollab's resources do not themselves possess classifications (the action parameter value is used to allow each resource to carry out various CRUD actions on a per-request basis), a similar analysis of WebCollab's most frequently improved functionality by classification was not possible based on coverage data. However, the three test suites for WebCollab all had "task" in common as the PSE of their most frequently improved resources. This again shows the trend that the PSE-Matching Algorithm retains some consistency in the modifications it makes across test suites. However, unlike the Logic results, the analysis of low-improvement resources across the three WebCollab suites yielded the prediction of "usergroup" as a potentially missed PSE. By knowledge of the web application, this potential PSE is indeed an object in persistent state that the algorithm is unable to associate with ID-Parameters due to internal naming inconsistencies.

6.4.4 Time Complexity of Classification Extraction and PSE-Matching Algorithms

Table 6.3 shows the time required by the classification extractor to determine classifications and request PSEs of all unique requests in the usage logs for each subject application. Because there are substantially more user session logs for Logic than for WebCollab, I include the calculation time for both applications.

Web Application	**Real Time (s)**	**User + System Time (s)**
Logic (626 sessions)	0.294	0.173
WebCollab (30 sessions)	0.203	0.129

Table 6.3: Time requirements for the Classification Extractor

The algorithm has $O(MN)$ time complexity where N is the number of requests in the test suite and M is the number of unique request combinations (i.e. combinations of resource name, request method, and parameter set) in the suite. This has an upper limit of $O(N^2)$ because the number of unique requests will be at maximum the number of total requests.

Test Suite Size (Number Cases)	**Real Time (s)**	**User + System Time (s)**
100	7.086	6.365
200	15.002	13.79
300	27.508	22.045

Table 6.4: Time requirements for the PSE-Matching Algorithm

Table 6.4 shows the time required by the PSE-Matching Algorithm to modify three Logic application test suites of varying size. The time complexity of the PSE-Matching Algorithm is $O(MN)$ where N is the size of the test suite in terms of number of requests and M is the number of elements in the Persistent State Map. This has an

upper limit of $O(N^2)$ since the size of the Persistent State Map will never exceed the number of requests in the suite.

6.5 Threats to Validity

In this section, I offer experimental details that may pose a threat to the validity of findings presented in this chapter.

To acquire usage logs for WebCollab and ORPM, I personally interacted with both applications deployed on a local server. My interactions with each application modeled the sessions of multiple different users and exercised as many usage patterns and functionalities as possible, however the data logged from these model sessions cannot be considered authentic usage data.

Although the data in the usage logs for WebCollab is not authentic, it succeeds in providing both valid and invalid parameter sets to be used in the PSE-Matching Algorithm and provides multiple parameter sets for each unique request combination. Therefore, the usage logs do not limit or inform the PSE-Matching Algorithm in any way that would increase or decrease the success of the PSE-Matching Algorithm more than authentic usage data likely would.

For the sake of the Parameter Naming Exploratory Study, the lack of authentic usage data poses less of a threat because the usage data was only a data set from which to mine as many ID-Parameters as possible. Note that there is no guarantee that the Logic application's usage data reveals all of the Logic application's ID-Parameters either, even though the usage data for Logic is data compiled from real user sessions.

6.6 Summary of Analysis and Observations

While the PSE-Matching Algorithm does, on average, secure some improvement in code coverage of applications under test, there is much room for improvement. The following insights follow from the data presented in this chapter:

1. While the PSE-Matching Algorithm achieves some success with the Persistent State Map's initial state containing no data, it is unlikely that this accurately represents a subject application's initial state when testing. Most often, subject

applications are tested after having their external data stores populated with some initial state. If the PSE-Matching Algorithm were to leverage the initial state of a subject application, this would likely create considerable improvements in coverage.

2. The PSE-Matching Algorithm is unable to account for internal application-specific dependencies. For example, for a series of requests to succeed, it may be necessary that a certain parameter value is held constant across each request. Dependencies such as this, or dependencies related to user permissions, etc., are difficult to detect without a knowledge of application-specific implementation details.

3. In Section 6.4.3, analysis of low and high- improvement Logic resources revealed that the PSE-Matching Algorithm improved no Read-classified resources. This is primarily because the Logic application contains fewer Read-classified resources than Update or Delete -classified resources for its most commonly accessed PSEs like "professor" or "student". However, I predict that Read-classified requests may, in general, rely on fewer internal dependencies to succeed. In general, the code for accessing an element's attributes just to display them likely does not contain as many executional branches as the code to modify an element– first checking to ensure the element exists, then checking permissions to allow editing, then performing validation of the new attributes provided for the entity, etc. Thus, if the parameter values passed into requests for an application *do* indeed decide the execution branches exercised by these requests, the fact that Update and Delete-classified resources saw more coverage improvement upon modification of these parameter values follows, along with the fact that Read-classified requests succeeded often enough by brute force pseudo-random parameter selection (in the test suites created with the Simple model) and thus experienced no improvement. This idea lends itself well to future study with additional web applications.

4. The PSE-Matching Algorithm operates on the premise that every CRUD-classified request it encounters as it traverses a test suite is *successfully* executed. It does not account for the possibility that some CRUD-classified requests may fail, such as requests attempted in a user session with an invalid login. Thus, the PS-Map begins to maintain state that does not accurately represent persistent state when it encounters many failed requests.

Chapter 7

CONTRIBUTIONS AND FUTURE WORK

In this chapter, I recount the contributions of the work I have presented and make recommendations for future work.

7.1 Contributions

The contributions of my work are:

1. **Language-based techniques for determining HTTP request's implications for persistent application state:** The exploratory studies presented in Chapter 3 provide evidence to support the existence of certain naming conventions that can be leveraged to predict the operations performed by HTTP requests.

2. **Dynamic prediction and tracking of elements in persistent application state throughout the traversal of a test suite:** The PSE-Matching algorithm presented in this thesis uses a simple map, the Persistent State Map, to model the data predicted to exist in persistent state as it traverses a test suite.

3. **Algorithm to apply persistent state predictions and create a persistent-state aware test suite:** Using the Persistent State Map, the PSE-Matching Algorithm provides request parameter values that create a greater coherence between test cases within a suite.

4. **Persistent-state aware test suite improves coverage for applications written in two different programming languages:** Comparison of code and branch coverage achieved by test suites before and after modification by the PSE-Matching Algorithm shows improvement in test suite performance for the Logic application written in Java and the WebCollab application written in PHP. This, along with the results of exploratory studies performed in Chapter 4.3, indicates that the persistent state mapping approach is extensible to a variety of web applications.

7.2 Future Work

The approach discussed in this thesis lends itself to further development in the following ways:

1. A study of fault types most and least frequently detected by the PSE-Matching Algorithm will likely reveal potential for specialization of the algorithm, if certain faults are revealed more than others. Evaluation of the PSE-Matching Algorithm's performance using a larger set of usage data as input will allow for a more thorough understanding of relationship between input size and algorithm success.

2. Evaluation of algorithm on additional subject applications will further test the extensibility of the algorithm.

3. Implementing algorithm improvements discussed in Section 6.6 would will likely improve coverage achieved by Persistent-State Aware test suites.

4. A study that analyzes most common usage patterns relating to entities in persistent state will likely be useful in further informing the PSE-Matching Algorithm's selection of ID-Parameters for requests. Because the PS-Map stores the most recent action performed on each PSE as well as its ID, an understanding of common usage patterns with regards to CRUD actions on PSEs could be leveraged in a useful way.

BIBLIOGRAPHY

[1] How to split text without spaces into list of words? http://stackoverflow.com/questions/8870261/how-to-split-text-without-spaces-into-list-of-words, 2012.

[2] Phpcoverage. http://phpcoverage.sourceforge.net/, 2013.

[3] Cobertura. http://cobertura.sourceforge.net/, 2012.

[4] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 49–59, May 2003.

[5] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user session data to support web application testing. *IEEE Trans. on Software Engineering*, 31(3), 2005.

[6] B. Korel. Automated software test data generation. In *Transactions on Software Engineering (TSE)*. IEEE, Aug 1990.

[7] Bigprof software. http://bigprof.com/appgini/applications/online-rental-property-manager, 2016.

[8] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Proceedings of the Automated Software Engineering Conference*, pages 132–141, September 2004.

[9] Jessica Sant, Amie Souter, and Lloyd Greenwald. An exploration of statistical models of automated test case generation. In *International Workshop on Dynamic Analysis (WODA)*, May 2005.

[10] Sara Sprenkle, Lori Pollock, and Lucy Simko. A study of usage-based navigation models and generated abstract test cases for web applications. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Mar 2011.

[11] Webcollab. http://webcollab.sourceforge.net/, 2016.