Utilizing Output in Web Application Server-Side Testing

Nadia Alshahwan

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

September 20, 2012

Declaration

I, Nadia Alshahwan confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Some of the work presented in this thesis has been previously published in the following papers [1]:

- Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pages 3–12, 2011.

- Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity (NIER track). In *Proceedings of the 34rd International Conference on Software Engineering (ICSE '12)*, pages 1345–1348, 2012.

- Nadia Alshahwan and Mark Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *the 21st International Symposium on Software Testing and Analysis (ISSTA '12)*, Pages 45–55, 2012.

The following papers, that are relevant to web application testing, have also been published during the program of study but do not appear in this thesis:

- Nadia Alshahwan, Mark Harman, Alessandro Marchetto, and Paolo Tonella. Improving web application testing using testability measures. In *Proceedings of 11th IEEE International Symposium on Web Systems Evolution (WSE '09)*, pages 49–58, 2009.

[1] Permission to reproduce these papers here has been granted by IEEE and ACM.

- Alessandro Marchetto, Roberto Tiella, Paolo Tonella, Nadia Alshahwan, and Mark Harman. Crawlability metrics for automated web testing. *International Journal on Software Tools for Technology Transfer (STTT)*, 13:131–149, April 2011.

- Nadia Alshahwan, Mark Harman, Alessandro Marchetto, Roberto Tiella, and Paolo Tonella. Crawlability metrics for web applications. In *Proceedings of the 5th International Conference on Software Testing, Verification, and Validation (ICST '12)*, 151–160, 2012.

To my Mother, Hussah Alodan, whose sacrifices, love and never ending support I could never repay.

Abstract

This thesis investigates the utilization of web application output in enhancing automated server-side code testing. The server-side code is the main driving force of a web application generating client-side code, maintaining the state and communicating with back-end resources. The output observed in those elements provides a valuable resource that can potentially enhance the efficiency and effectiveness of automated testing. The thesis aims to explore the use of this output in test data generation, test sequence regeneration, augmentation and test case selection.

This thesis also addresses the web-specific challenges faced when applying search based test data generation algorithms to web applications and dataflow analysis of state variables to test sequence regeneration.

The thesis presents three tools and four empirical studies to implement and evaluate the proposed approaches: SWAT (Search based Web Application Tester) is a first application of search based test data generation algorithms for web applications. It uses values dynamically mined from the intermediate and the client-side output to enhance the search based algorithm. SART (State Aware Regeneration Tool) uses dataflow analysis of state variables, session state and database tables, and their values to regenerate new sequences from existing sequences. SWAT-U (SWAT-Uniqueness) augments test suites with test cases that produce outputs not observed in the original test suite's output. Finally, the thesis presents an empirical study of the correlation between new output based test selection criteria and fault detection and structural coverage.

The results confirm that using the output does indeed enhance the effectiveness and efficiency of search based test data generation and enhances test suites' effectiveness for test sequence regeneration and augmentation. The results also report that output uniqueness criteria are strongly correlated with both fault detection and structural coverage and are complementary to structural coverage.

Acknowledgements

First, I would like to thank My supervisor Professor Mark Harman who encouraged me to undertake this PhD and taught me a great deal about research and academia. I also want to thank him for his continuing advice, support and patience throughout my PhD, and for the countless opportunities he offered me over the years. I would also like to thank my sponsors, the ministry of higher education in Saudi Arabia, for making this PhD possible. I would also like to thank my co-authors, and in particular Professor Paolo Tonella, for their hard work. I want to thank my wonderful colleagues at the CREST centre, especially Kelly Androutsopoulos and Bill Langdon, for the countless interesting discussions and advice on writing up this thesis. I want to thank Eric Bouwers, from Delft University of Technology, for making the static analysis library PHP-Front, which is used throughout this thesis, available and never failing to respond to my emails. Last but not least, I would like to thank my parents, my best friend, Puris Fazie, and the rest of my family for their continuing love, understanding and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A web application is a special client-server software system that is available through the internet and executes within a web browser. The server-side code implements the business logic of the application while the client-side code provides the interface to the user. The client-side and server-side communicate using a transfer protocol, typically HTTP (Hypertext Transfer Protocol). The client-side page can be static or dynamically generated by the server-side code based on user choices.

The importance of automated web application testing derives from the increasing reliance on these systems for business, social, organizational and governmental functions. Over the past ten years, internet user numbers grew by approximately 528% [Sta11]. In 2009, online retail sales grew by 11% compared to 2.5% for all retail sales [Tec10]. Amazon, the leading online retailer, increased its sales by 29.5% [Ret09].

One of the advantages of web applications is their continual availability. The service provided by a web application is not limited by location or time, since geographically separated users may have concurrent access. However, these advantages impose a demand for high availability.

Web time is considered to be 5 to 7 times faster than normal time [Ger00]: Web technologies change more frequently and their adopters seek early acquisition of market share. This pressure on development time squeezes the testing phase, especially when it is unautomated, labour intensive and, therefore, slow.

However, inadequate testing poses significant risks: Studies showed that trust and convenience are major factors affecting customer loyalty using web applications [AS03]. Both recent and historical studies have shown that online shoppers exhibit impulsive purchasing habits [BKMD09, DA99], indicating that downtime can prove

costly. For example, downtime was estimated to cost Amazon $25k per minute even as early as 2001 [PN05].

The critical need for high availability and the short development lifecycle of web applications necessitate the development of sophisticated automated testing techniques. On the other hand, the unique nature of web applications could be exploited to enhance the testing process.

The output of the server-side code of a web application is rich and complex compared to traditional applications: The client-side page is structured and full of useful information in `forms` and embedded links. A web `form` in the client page can contain values in drop-down menus, lists and default values of fields. In addition to the client-side page, the output can also be observed in changes in the application's state, such as the database or the server session state. Although parts of the output have been utilized successfully in other domains, such as Graphical User Interface (GUI) testing [YM10, YM07], little work [ECIR06, JHHF08, SBV+08] attempts to take advantage of the output of web applications to enhance testing of web applications.

In this thesis, the term output is used to describe any values that are calculated and produced by the system under test whether these values can be observed at an intermediate point or at the end of the program execution. That is, in addition to the client-side page produced by a web application's server-side code, changes to state variables, SQL commands produced by the program and sent to the SQL server as well as values of variables at an intermediate point in the program are considered output.

The thesis investigates the way in which different parts of the output can be used to enhance testing of the server-side code of web applications: The values in an output client-side page can be mined and seeded into a test data generation process. Changes in the server-side state, when executing a test case, can be tracked to provide insight into the application's behaviour. This insight can then be exploited in constructing interesting test sequences that are more effective in finding faults and exploring new parts of the application. Moreover, because the client page is rich and structured, changes in that page in response to different input values could indicate interesting executions of the application and could, therefore, act as test adequacy/selection criteria.

The next section provides further details about the problems addressed in this thesis and the techniques proposed to solve them.

1.1 Problem Statement

To test the server-side code of a web application, two layers of testing have to be considered [TR04a]: Code level structural testing and higher level navigational testing. Code level testing is similar to traditional application testing and requires the generation of test data that aim to maximize structural coverage. Navigational level testing requires the construction of sequences of requests that exercise the application's behaviour. These two types of testing are not independent because generating test sequences requires the individual requests in each sequence to have effective test inputs. On the other hand, sequences of requests might be needed to cover a branch that requires the application to be in a specific state.

The thesis makes use of search based approaches in test data generation. Search based approaches formulate software engineering problems into optimization problems. For test data generation, a random test case or test cases are first generated and executed. During execution, a fitness function is used to measure how close each test case came to covering the target branch. This fitness function provides guidance to the search algorithm in adjusting the values of inputs until the target branch is covered. Test data generation remains one of the dominating applications of search based techniques in software engineering. Surprisingly, search based approaches have not been applied yet to web application test data generation. Web applications' reliance on strings and the flexible and dynamic nature of their implementation languages pose new challenges to search based techniques.

Previous work on search based test data generation seeded values from various sources into the search space to replace the randomly generated initial test case(s) thereby enhancing effectiveness, by achieving higher coverage, and efficiency, by covering branches faster. These seeding sources included constants collected from the source code [AB06], test inputs that were collected while attempting to cover other branches [MMS01] or test cases from previous releases or manual testing [YH10]. The results of the evaluation of these approaches suggest that seeding can potentially enhance the performance of search based approaches. The output of a web application can be a new source of input values that can be used for seeding.

The approach presented in this thesis for test data generation and implemented in the tool SWAT (Search based Web Application Tester), applies search based algorithms

to test data generation by dealing with challenges that are specific to web applications. The approach also investigates using the client-side and intermediate output of a web application as a source of seeding and its effect on the performance of the approach.

When testing a web application on the higher navigational level, previous work has focused on generating a model of the application that is then used to generate test sequences that cover the model satisfying some tester chosen criteria [RT01a, KT01, QMZ07, RT02]. Using this approach to generate test sequences raises three key issues:

1. The test inputs attached to each request are left to be inserted manually by the tester.
2. The number of test sequences that are needed to cover even a simple model can be large especially with the presence of cycles.
3. The generated models only represent the developer's intended use of the application since they are typically constructed by analyzing links and `form` submissions between pages. In practice, the user can follow any path through the application since web browsers provide users with the ability to access any desired page directly (through the address bar or bookmarks) and to alter an execution path by using the `Back` and `Forward` buttons.

Client-side pages impose restrictions on how the web application is used: validations on fields in web `forms` implemented using client-side scripts (e.g., JavaScript) prevent invalid input values from being passed to the server. Navigational menus and links impose restrictions on the execution flow of the application. However, in practice the user can bypass these restrictions intentionally or unintentionally by, for example, disabling client-side scripting languages or altering the intended execution flow by using a browser's navigation buttons (e.g., `Back`, `Forward`) or address bar. This ability to bypass client-side restrictions emphasizes the importance of testing the server-side code comprehensively without limiting the testing process to scenarios that are defined by the client-side code.

The approach proposed in this thesis for generating sequences attempts to address these issues: It generates sequences that are not restricted by the application's model and presents an algorithm that generates limited test sequences that are more likely to cause the application to explore new behaviour. This new algorithm uses dataflow analysis of the web application's server-side state variables to generate test sequences

that cover Def-Use pairs of those variables. The main argument behind this approach is that HTTP is stateless; therefore the state (represented in session variables and database tables) is the only element that propagates over a sequence of requests. The approach also reuses the test data generated by SWAT to eliminate the manual effort needed from the tester while using potentially effective test data that maximizes coverage.

Many studies have investigated the effect of maximizing structural coverage on test suites' effectiveness in fault detection [ABLN06, HFGO94, FW93, NA09]. The results of those studies showed that, although higher structural coverage is correlated with a test suite's effectiveness, it is not the only contributing factor. For example, Hutchins et al. [HFGO94] reported that achieving 100% structural coverage of a system under test does not guarantee finding all faults.

This thesis proposes that the diversity of the output observed by executing a test suite can be an indication of a test suite's effectiveness in detecting faults. Therefore, the effectiveness of using output as a test adequacy/selection criterion is investigated as a potential complementary criterion to structural coverage. The correlation between output diversity and structural coverage is also investigated to explore the possibility of using output as a stand-alone criterion when structural coverage cannot be used. For example, when all or part of the code is unavailable for instrumentation.

1.2 Objectives

The objectives of this thesis are the following:

1. Applying search based test data generation algorithms to web applications while extending them to handle web-specific challenges and to utilize output in the search process. Search based techniques have been shown to be complementary to Dynamic Symbolic Execution approaches [LMH10] which have been previously applied to web applications [AKD+08, AKD+10]. Constant seeding approaches for string predicates have been shown to enhance search based techniques' effectiveness and efficiency [AB06]. Since web applications are string based, it is expected that constant seeding will have the same beneficial effect when applied to web applications. However, utilizing the output by dynamically extracting values and seeding them into the search space might result in a greater improvement on the test data generation process.

2. Investigating the use of dataflow analysis of the application's server-side state in test suite regeneration. Since HTTP requests are stateless, the server-side state maintained in session variables and database tables is mainly what propagates over a sequence of requests to the web application server. Therefore, to generate sequences that are effective in increasing coverage and finding faults without exhaustively trying every possible sequence, which might be infeasible even for small applications in the presence of cycles, analyzing the state could prove to be effective in generating those sequences.

3. Investigating the effectiveness of using output uniqueness as a test selection criterion for test suite augmentation. In applications with rich outputs, such as web applications, the possibility of faults propagating to the output could be high. The uniqueness of an output might indicate an interesting test case that could reveal faults even when traditional structural coverage criteria have not been improved by the test case.

4. Empirically investigating the correlation between output uniqueness and test suite effectiveness as well as output uniqueness and structural coverage. The goal of the empirical study is to explore the effect of the output on fault finding ability and compare it to structural coverage criteria. Moreover, a strong correlation between output uniqueness and structural coverage can lead to a test data generation approach that is based on output uniqueness when structural coverage cannot be obtained.

1.3 Contributions

The contributions of this thesis are:

1. An automated tool SWAT (Search based Web Application Tester) for PHP web applications test data generation.

2. The introduction of a dynamic seeding approach that uses the application's output to enhance search based test data generation.

3. An empirical evaluation of the output utilization enhancement to search based test data generation techniques that are introduced in SWAT. The evaluation con-

firms that the enhancement improves both effectiveness and efficiency of the generation process.

4. An automated approach for test suite regeneration that generates test sequences from an existing test suite based on dataflow analysis of the web application state variables and their values and a tool SART (State Aware Regeneration Tool) that implements the approach.

5. An empirical evaluation of the dataflow based test regeneration approach that confirms that both coverage and fault finding ability can be enhanced using this approach.

6. The introduction and evaluation of novel test adequacy criteria based on output uniqueness together with seven definitions of output uniqueness.

7. A case study and empirical study that verify the usefulness of the new output uniqueness criteria in test data selection and augmentation.

1.4 The Structure of the Thesis

Chapter 2 provides a review of the literature that is most relevant to this thesis. The chapter describes web applications' architecture and discusses testing challenges that are specific or more prominent in web applications. It reviews previous research relevant to the main themes of this thesis: web application testing, search based test data generation, test adequacy criteria and the use of output to enhance testing.

Chapter 3 introduces three related seeding algorithms and a tool, SWAT, for automated web application testing using Search Based Software Testing (SBST). The algorithms significantly enhance the efficiency and effectiveness of traditional search based techniques exploiting both static and dynamic analysis. The combined approach yields a 54% increase in branch coverage and a 30% reduction in test effort. Each improvement is separately evaluated in an empirical study on six real world web applications. The work presented in this chapter is based on a paper published by the author of this thesis in ASE in 2011 [AH11].

Chapter 4 introduces a test case regeneration approach for web applications that uses a novel value-aware dataflow analysis approach. The overall approach is to combine

requests from a test suite to form request sequences, based on dataflow analysis of server-side session variables and database tables. The chapter also introduces a tool SART (State Aware Regeneration Tool) that implements the approach and an evaluation of the effectiveness of the approach in enhancing branch coverage and fault detection. The work presented in this chapter is based on a paper published by the author of this thesis in ISSTA 2012 [AH12b].

Chapter 5 introduces novel test adequacy criteria that are based on the HTML output of web applications. Seven definitions of output uniqueness are proposed that have varying degrees of strictness. The chapter presents a case study that investigates the effectiveness of using the new criteria in test suite augmentation. The study has been previously published by the author of this thesis in ICSE New Ideas and Emerging Results (NIER) track [AH12a]. The chapter also presents an empirical study that investigates the relationship between the new output based criteria, existing structural criteria and fault detection ability of test suites.

Chapter 6 summarizes the conclusions derived from the work presented in this thesis and describes plans for future work.

1.5 Applications Studied and Oracle

This Section describes the web applications and fault detection oracle that are used in experiments throughout this thesis. These web applications were chosen as evaluation subjects because they are used by real users, represent different domains and were used in previous studies by other authors for web application testing [AKD+08, AKD+10, Min05]. The fault detection oracle is one that is fully automated thereby eliminating experimenter subjectivity when deciding what is counted as a fault to evaluate an approach's fault detection capability.

1.5.1 Applications Studied

The web applications that are chosen as test subjects are all implemented in PHP. PHP is one of the top rated open-source server-side scripting languages [TIO12]. PHP was chosen because it has many features that pose interesting challenges when implementing automated testing approaches, such as dynamic typing and dynamic file inclusions.

All applications are open-source and available to download on the SourceForge[1] web page. The applications are used by real users as indicated by the number of downloads reported by SourceForge which range from 13k to 774k downloads. For example, PHPBB2 has many versions and supporting files and themes that are available through SourceForge but also has a supporting community and independent web page.[2]

Table 1.1 provides details about the applications selected. For each application, the version that was used together with information about the size of the application and a brief description are provided. The applications range in size from 800 to 22k executable PHP Lines of code and consist of between 19 and 78 PHP files. Other file types, such as static HTML and JavaScript files were not counted as the approaches presented in this thesis are related to server-side PHP code. All applications except PHPSysInfo use a database and require user authentication to access all or part of their functionalities.

Table 1.1: Subject web applications

App Name	Version	PHP Files	PHP ELoC	Description
FAQForge	1.3.2	19	834	FAQ management tool
Schoolmate	1.5.4	63	3,072	School administration system
Webchess	0.9.0	24	2,701	Online chess game
PHPSysInfo	2.5.3	73	9,533	System monitoring tool
Timeclock	1.0.3	62	14,980	Employee time tracking application
PHPBB2	2.0.21	78	22,280	Customisable web forum

1.5.2 Fault Oracle

Determining the fault finding ability of approaches is necessary in comparing their effectiveness. However, the determination of whether or not an approach has found a fault needs to be free from experimenter bias and subjectivity. Therefore, an automated oracle is used throughout this thesis when evaluating the fault finding ability of each proposed approach.

[1] http://sourceforge.net/

[2] http://www.phpbb.com/

The automated oracle parses PHP error log files and the output HTML page of each test case for faults. Only faults that are caused by a unique code location and have a distinct type are counted (to avoid double counting of faults). If two distinct faults appear in the same statement but have the same type, only one fault is counted. This method of distinguishing unique faults is used to keep the oracle completely automated: Deciding whether two faults of the same type that appear in the same statement but have different error messages denote distinct faults requires manual intervention.

The errors reported in the PHP error log files are of four types: errors, warnings, notices and strict warnings. Faults indicated by the log file to be of type 'error' are faults that cause the execution of scripts to abort. This can happen, for example, when non-existent files are included or functions are called that are not found. Therefore, these fault types can be classified as crashes.

Faults reported as PHP 'warnings' are defined in the PHP manual as non-fatal errors.[3] An example is the execution of an SQL statement that is invalid. Although these faults do not cause the application to crash (due to the fault-tolerant nature of PHP), they could indicate vulnerabilities and bugs in the code that need to be addressed. These fault types together with those parsed from the output HTML are classified as errors. The faults parsed from the output HTML are SQL faults that cause the execution of the PHP script to terminate while printing an error message to the user.

PHP notices and strict warnings are usually regarded as less serious and, therefore, are classified as warnings. An example of this fault type is using a variable or index that has not been defined.

[3] `http://www.php.net/manual/`

Chapter 2

Literature Review

This chapter reviews the literature that is relevant to this thesis. First, a description of web applications and a discussion of web testing challenges that are identified in the literature are presented. Then, the main approaches proposed by research on web application testing are reviewed. A background on Search Based Software Engineering and the approaches and concepts that are currently used in search based test date generation are provided. This chapter also reviews test adequacy criteria used in both traditional systems and web applications. Finally, previous work that utilizes output in enhancing the testing process is reviewed.

2.1 Web Application Architecture

The architecture of a typical dynamic web application (Figure 2.1) consists of three tiers: the presentation tier, the application tier and the data tier.

The presentation tier is the client-side pages that the user sees when accessing the application using a browser. These can be static HTML pages or dynamic pages generated by the server. In addition to the content and HTML code, these client-side pages can include scripts, such as JavaScript or VBScript, plug-ins, such as Flash and Java applets, and Cascading Style Sheets (CSS) that format the output. Client-side scripts perform actions on the client-side, such as input validations, without communicating with the server.

The application tier contains the server-side code and is invoked by the presentation tier through HTTP requests, such as `form` submissions or direct links. The server-side code processes user requests and generates new client-side pages based on user choices.

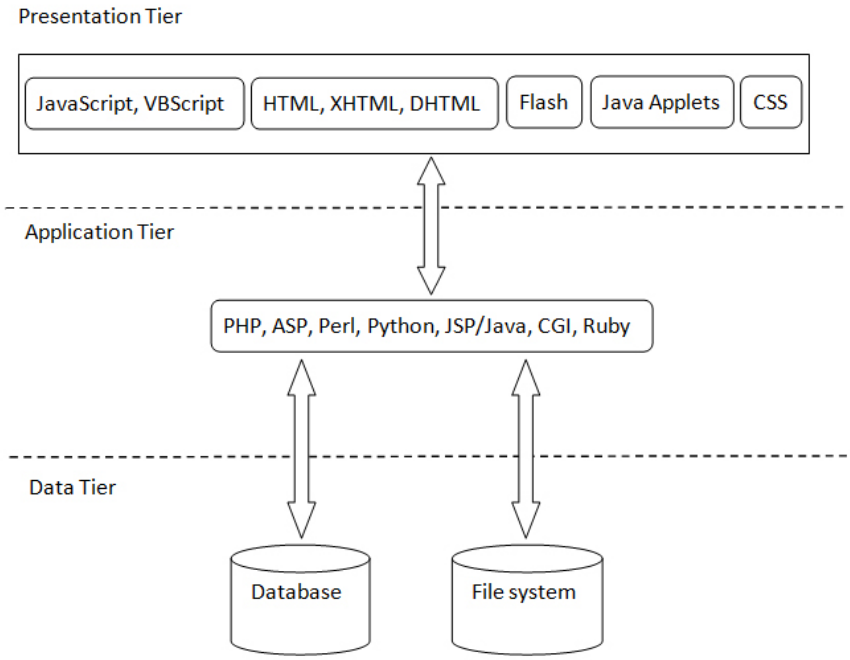


Figure 2.1: Typical web application architecture

The data tier could be a file system or, more typically, a database that is accessed by the server-side code when needed. The database holds information previously saved by the user on previous requests (e.g., user profile in a social network) and/or data uploaded by the web application's administrator (e.g., product information in an e-business application).

2.2 Web Testing Challenges

The testing of web applications shares many challenges with testing of traditional applications but also poses some challenges that are unique to web applications [BFG02, WO02].

Web applications differ from traditional applications in that they are globally available to any person (or script) that uses the internet. The number of users accessing the application is unlimited and could change unexpectedly. A significant increase in the number of users could affect the performance of the application. No assumptions about the experience levels of users can be made, resulting in unexpected usage patterns that might cause invalid application behaviour.

The development lifecycle of web applications is much faster than traditional applications [Ger00]. The pressure caused by those frequent changes leaves limited time for testing and verification of the application. This highlights the need for automated testing tools and approaches.

As web applications typically communicate with many different components, such as the database, file system, CGI scripts and external web services, these communications affect the testing process and need to be addressed.

The APIs of web applications are not explicitly defined, making automated generation of test data more challenging. The client and server interact using global arrays (e.g., `$_POST`, `$_GET`) that are set on the client-side and accessed by the server-side at any point in the program. This also causes web applications to be vulnerable to misuse and attacks, since the design of these global arrays imposes no restrictions on the number, names and types of input values. Even when client-side scripts are used to validate the values provided by the user, the ability to bypass the interface (for example, by disabling JavaScript) leaves the application vulnerable to attacks.

As discussed in the previous section, web applications are typically composed of server-side and client-side code. Client-side pages can be static but are more typically generated dynamically by server-side code based on user choices. These dynamic pages can contain client-side scripts that manipulate the interface or perform input validations as well as links and `forms` that interact with the server-side code. Because this dynamic content affects how the application behaves, understanding a web application's behaviour and the connections between its components might only be possible at run-time. Moreover, the interactions of server-side and client-side code need to be tested as well as testing server-side and client-side code separately.

Web applications are typically rendered in a browser. Using browser functions, the user has extended control over the execution of the application. This makes it possible to enter the application at any point since the user can type a specific URL directly into the address box of the browser or click a previously saved bookmark. The execution flow can also be altered by the user by clicking the `Back`, `Forward` or `Refresh` buttons on the browser. These browser functions provide the user with the ability to alter the execution paths intended by the application's developer causing unexpected behaviour that might cause the application to fail [DLDP03]. Empirical investigation of

user action logs (session logs) revealed that these altered execution paths are common [TR04b].

The nature of HTTP requests is stateless; the server processes each request in isolation without any knowledge about previous requests from the same user. Other methods, such as sessions and cookies are used to propagate state; this makes the behaviour of the application dependent on factors, such as sessions timing-out or cookies being enabled.

The environment in which web applications run is neither well-defined nor static. Developers have no control over the type and version of browser used, the configurations of the user machine, browser or the connection type. The content of the web application could be rendered differently in different browsers [MP11, RCVO10, EM07]. Browser configurations also affect the application behaviour by, for example, disabling cookies or JavaScript.

2.3 Web Application Testing

Early research on web application testing identified the differences and similarities between web and traditional software testing. The testing processes that are defined for traditional software were amended to be applied to web application testing. Yang et al. [YHWC99, YHW+02] proposed a web testing architecture that is modelled on traditional testing architectures but adapted for the unique characteristics of web applications. For example, a new process was added to the traditional architecture to address the mixed programming nature of web applications; the same page can contain several programming languages (e.g., HTML, JavaScript, VBScript). Moreover, all other testing processes' internal specifications were adapted to the web. When analyzing failure, for example, Yang et al. suggested that the output URL, `form` fields and graphical output need to be checked.

Research on web application testing in the early 2000s focused on applying model-based testing techniques to web applications. This was caused by the introduction of dynamic web applications in the late 90s and the seminal work on model-based testing of web applications by Ricca and Tonella [RT00, RT01a, RT02]. In many of those approaches, a model of the application is constructed and then used to generate test sequences that exercise the model and achieve a tester specified coverage

criteria. Research also focused on advancing the state-of-the-art of web crawlers [BFG02, RGM01]. In 2003, Elbaum et al. [EKR03] proposed using session data saved in the web server's logs to generate test suites that are more realistic and reflect actual user behaviour. This approach was subsequently developed by other researchers [SMSP04, AH08, SGSP05b, LPC09]. At the same time, research developed that focused on certain characteristics that are specific to web applications, such as browser interactions [DLDP03, LK04] or the user's ability to bypass the interface [OWDH04a, OWDH04b]. Minamide [Min05] presented a method for fault detection in PHP applications by statically approximating the output. In 2006 attention focused on identifying the interfaces (the actual input fields and their expected values or ranges) that a web application uses [ECIR06, HO07]. This led to subsequent work that focused on generating test data that satisfied some white box test adequacy criteria [AKD+08, WYC+08]. Figure 2.2 depicts a timeline of developments in web testing research.[1]

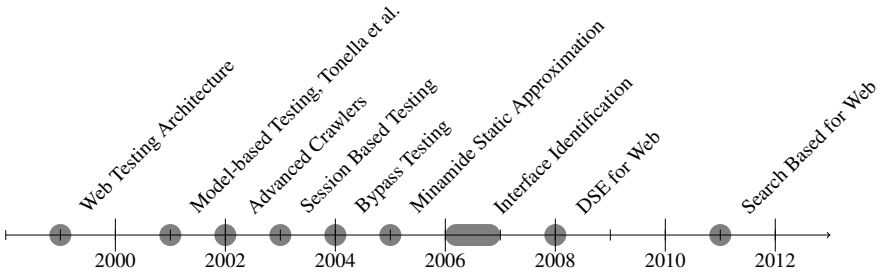


Figure 2.2: Major milestones in web application testing research

Di Lucca and Fasolino [DLF06] examined and summarized the state-of-the-art in web application testing and suggested future trends. They concluded that to apply traditional testing techniques to web applications, considerable effort in adapting these techniques to web applications is needed when the testing process is dependent on implementation. This is due to the specific characteristics and dynamic nature of web applications.

The remainder of this section briefly examines available testing tools that are currently used in practice and reviews the main web testing approaches.

[1] Search based for web is the work of the author of this thesis [AH11]

2.3.1 Available Tools and Techniques

Due to the importance of high availability, scalability and security for web applications, the focus on non-functional testing practices is extremely high. Tools that perform load and security testing are used regularly to ensure that a web application meets these requirements. For functional testing, available tools range from HTML validators and link checkers to capture/replay tools and testing frameworks that aid the tester in writing testing scripts to perform unit testing. A considerable amount of manual effort is still required in using these tools.

Crawlers

Web crawlers can be viewed as testing tools as they traverse a web application reporting failures and broken links. Many open-source and commercial crawlers exist with varying degrees of functionalities. WebSPHINX [MB98] is a customizable site-specific spider with a GUI interface. It also provides a class library that can be used to implement customized spiders in Java. Mercator [HN99] has similar features to WebSPHINX but is designed to deal with scalability issues. JSpider [Jav03] is another open-source crawler that has the advantage of recording a web application's structure to a database. However, all three crawlers do not provide support for automated `form` filling and submission. Teleport Pro[2] is a commercial crawler that provides a few additional features: The crawler gives the user the ability to provide authentication information to access password protected parts of the application. It also parses JavaScript to extract links. Girardi et al. [GRT06] conducted a comparison of these and other crawlers based on completeness, robustness, features offered and download limiting options. The study concluded that the considered crawlers have different strengths and weaknesses but indicated that some commercial crawlers offer more completeness. Although crawlers have the advantage of being fully automated, their ability to cover an application is closely related to their capabilities.

HTML Validators and Link Checkers

Link checkers[3] and HTML validators are a class of automated tools that are similar to crawlers. Link checkers go through all links in a web site or a page to check for

[2] http://www.tenmax.com/

[3] http://validator.w3.org/checklink

any broken links. They operate in a way somewhat similar to a search engine. Some of these tools have the additional capability of checking images and links contained in JavaScript. HTML validators check the HTML code of a page or web site for mismatched tags and other violations of HTML standards. Many browsers are able to display the page correctly even when HTML standards are not met. However, that is not always the case and it is better practice to ensure that HTML is formed correctly. One limitation of HTML validators is that they are only able to check static HTML pages. Several tools are available to check links or validate HTML both on–line and as downloadable executables. The World Wide Web Consortium (W3C) offers both types of tools on their web page.[4]

Capture/Replay Tools

Automated functional testing of web applications is mainly based on capture/replay tools. Existing capture/reply tools, such as LogiTest [Too01] and Maxq [OP09] provide the tester with a browser–like interface. The tester then navigates through the web application recording various testing scenarios which are then repeated during regression testing. However, the quality of the produced test suite relays on the tester's thoroughness and skill and requires significant manual effort. Also, changes in the structure of the web application could make the recorded test suite fail to run. This makes it necessary to re-record all or part of the tests. Selenium is a popular open-source capture/reply tool that also provides the ability to modify test scripts. Meszaros [Mes03] discussed the advantages of using capture/replay tools compared to crawlers and presented a simple implementation that incorporates capture/replay in an existing system.

Testing Frameworks

A number of testing frameworks are available to create unit tests for web applications. HttpUnit [Gol08] is a Java API that provides the building blocks required to emulate the browser's behaviour. When combined with a framework, such as JUnit,[5] HttpUnit allows testers to create test cases to verify web Application behaviour [HM02]. SimpleTest [Sim12] is a similar framework for PHP that additionally provides the ability

[4] http://www.w3.org/

[5] http://www.junit.org/

to create mock objects that simulate time-consuming operations, such as database connections. PHPUnit[6] is part of the xUnit family of testing frameworks. It provides comprehensive functionalities to create and run unit tests by utilizing tools, such as JMock[7] to create mock objects and Xdebug [Ret12] to analyse coverage. Testing frameworks provide an aid to the tester to create unit tests; however the design and creation of tests is manual and requires time and resources that might not be available or cost effective.

2.3.2 Testing Approaches

This section reviews the main web testing approaches proposed in the literature. It also reviews analysis approaches that can aid the test generation process, such as interface identification.

Model Based Testing

A web application consists of a collection of static or dynamic pages that are connected by links or `form` submissions. These pages are usually well connected to facilitate navigation and make the application more user friendly. The user of the application can follow different paths through the application, some of which might be unexpected, which makes it necessary to test web applications on a higher navigational level as well as the traditional code level [TR04a]. Model based testing of web applications addresses this navigational type of testing.

Modelling web applications and using the model to generate test sequences was initially proposed by Ricca and Tonella [RT01a]. Two tools were developed to support the approach: ReWeb and TestWeb.

ReWeb was initially introduced in 2000 [RT00] and was only able to analyse static applications. The tool constructed a model of the application for the purpose of aiding maintenance. ReWeb downloads a web application through crawling and constructs a model where nodes are pages and edges are links. The tool was used to download an application at different points in its lifetime and analyse the changes. The results are also presented graphically and include additional analysis information, such as shortest path to a page or connectivity of pages. ReWeb was later used to help in restructuring web applications [RT01c] to enhance reachability of pages or merge redundant menus.

[6] http://www.phpunit.de/

[7] http://www.jmock.org/

When applied to testing [RT01a, RT02], ReWeb was enhanced to handle `forms` thereby providing support for dynamic applications. ReWeb was enhanced to represent conditional links which are caused by a `form` submission that can lead to different pages depending on the user input. This enhanced ReWeb analyses a web application semi-automatically and produces a UML (Unified Modeling Language) model that represents the application. Figure 2.3 [8] demonstrates the meta-model of the generated UML model. A concrete UML model represents a web application using a combination of the elements in the meta-model. However, manual effort is required by the tester to provide input values for `forms` and define the required conditions on edges together with values that satisfy them. The model is then used by TestWeb to generate test suites that cover the model. The resulting test suite consists of HTTP request sequences with the input values left empty to be entered later by the tester. The completeness of the model is not guaranteed since it depends on the values provided by the user for conditional branches which can reduce the effectiveness of the generated test suite. For large applications, the conditions that could affect the resulting output page can be very complex and hard for the tester to specify manually. In essence, these conditions correspond to the complexity of the server-code and the number of branching statements it contains. More details about ReWeb and TestWeb and challenges in building these tools together with the used solutions are described in detail in the literature [RT01b].

However, even for a complete model that represents the application accurately, the possible sequences of requests that adequately cover the model can be very large in a more mature and complex application. Time and resources dedicated to testing are often limited. To overcome this problem Kallepalli and Tian [KT01] proposed modelling the usage patterns of a web application to help in performing statistical testing. The approach aims to identify the most frequently used sections of a web application by examining traces so that testers can distribute testing effort in a more effective way. Statistical testing prioritizes parts of the application that are more frequently used by the application's users. However, this does not eliminate the need to test other parts of the application: Web applications are available globally to users with different backgrounds and experience levels therefore usage can change unexpectedly at any time.

Qian et al. [QMZ07] proposed generating a simplified tree from the application's

[8] © 2001 IEEE. Reprinted, with permission, from [RT01a]

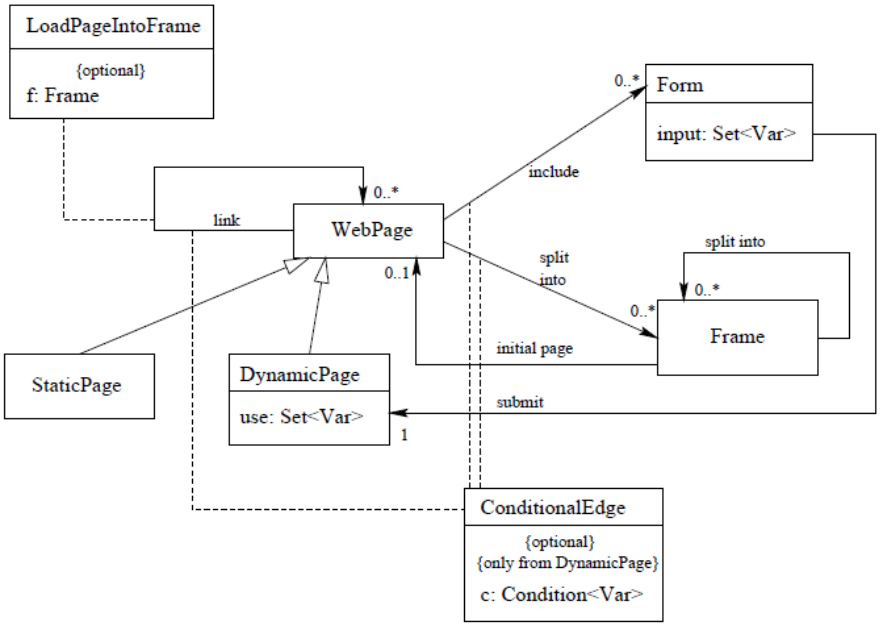


Figure 2.3: Meta-model of the UML models generated by ReWeb Taken from Ricca and Tonella's paper[RT01a]

graph to limit the number of sequences that need to be generated to cover the model. This approach provides an algorithm to generate a tree that covers all pages and all links in the application's model. The generated test sequences therefore would not satisfy more demanding criteria for covering the model, such as all paths. Their approach still requires manual intervention for entering values for user inputs and no evaluation was provided to analyse the impact of the simplification algorithm on effectiveness.

In model based testing approaches of web application, the models are generated dynamically by crawling the application. Therefore, for dynamic applications the model might not be complete since the possible different pages depend on the inputs that were used during the analysis. Wu and Offutt [WO02] introduced a novel approach that models static and dynamic aspects in web applications. For every file in the application, atomic sections are identified. An atomic section is a static HTML file or a part of the server-side code that prints HTML as a block. For example, statements printing HTML inside the `true` branch of an `if` statement are considered an atomic section since if the `true` branch is executed the whole section will be part of the generated client-side page. Figure 2.4 and Table 2.1 demonstrate the approach by showing a server-side script together with its atomic sections. Possible combinations of these

atomic sections are modelled using sequence, selection, and aggregation. Transitions between server and client-side components are also modelled. Test suites are then derived from the resulting model. The approach has not been automated and the models are still preliminary and for a large application could be very complicated limiting the scalability of the approach.

```
<?
 Print "<html>";
 Print "<body>";
 $num = $_POST['num'];

 if ($num/2 ==0)
   Print "$num is even!";
 else
   Print "$num is odd!!";

 Print "</body>";
 Print "</html>";
 ?>
```

Figure 2.4: Simple server-side script to demonstrate modelling static and dynamic aspects of web applications in Wu and Offutt's [WO02] approach.

Table 2.1: Atomic sections of the code in Figure 2.4. Each set of print statements that are printed together forms an atomic section.

1	Print "<html>"; Print "<body>"; $num = $_POST['num'];

2	if ($num/2 ==0)
	Print "$num is even!";
	else

3	Print "$num is odd!!";
4	Print "</body>"; Print "</html>";

Di Lucca et al. [DLFF02] developed a testing strategy for unit and integration functional testing of web applications. The paper proposes an object-oriented model of the web application as a testing model and uses a tool WAT (Web Application Testing) to support the approach. Single web pages (either client or server) are considered as units when performing unit testing. Stubs for all other components and a driver to execute the tests are generated to support the testing process. For integration testing, all pages related to a single use-case are tested together. The grouping of objects is done dynamically and incrementally until all related pages are included. The approach and tool aim to automate part of the testing process and aid the tester in making decisions. However, considerable manual effort is still needed, such as providing values for test inputs and creating the stubs. Although the results of a case study indicate that the approach could be effective, the evaluation is preliminary and only includes one application in ASP. The WAT tool uses the static analysis results of the tool WARE (Web Application Reverse Engineering) [DLDB06, DLFT04, DLFP$^+$02]. WARE is a static multi-language code parser that performs lexical and syntactical analysis to extract information about the application. The goal of the tool is to build conceptual models and use-case diagrams of the application.

With the dynamically generated pages of a web application, the dynamic modelling approaches proposed so far could result in very huge models with nodes that are slightly different but represent the same conceptual page. Andrews et al. [AOA05] partitioned the application pages into clusters to identify those pages that are conceptually the same but differ slightly in content. Each cluster is then modelled into Finite State Machines (FSMs) and an application FSM is built for the full system which is then used to generate test suites. The approach is not fully automated, but provides an editor that helps the tester in the process of generating tests.

One of the issues in web application testing is the effect of the database state on the test generation process and results. To eliminate the database effect the state can be initialized before every test case is executed. However, coverage could depend directly or indirectly on values retrieved from the database. This suggests that it could be beneficial to include the database in the test generation process. This issue is not specific to web applications since traditional applications can also use a database. However, the use of a database is widely spread in modern web applications.

Deng and Wang [DFW04] apply the AGENDA tool [CDF+04] to web applications developed in Java Servlets. Their approach uses static analysis to build a model of the application where nodes are URLs and edges are links. The approach classifies URLs into static and dynamic URLs based on database access. The model and the AGENDA tool are then used to generate test cases and inputs for interesting paths in the application. The static nature of the analysis limits the accuracy and completeness of the model: Web applications are hugely dynamic; URLs and input names can be constructed dynamically which static analysis is unable to detect. The AGENDA tool also requires considerable manual intervention from the tester in providing 'sample value files' that are used to populate the database.

Recently, model based techniques have been applied to AJAX web applications. AJAX (Asynchronous JavaScript And XML) is a new technology that allows implementation of rich and dynamic web interfaces by using asynchronous communication between the client and server. Marchetto et al. [MTR08] proposed an approach to test these applications by abstracting the Document Object Model (DOM) into a state model. The state model is constructed using static and dynamic analysis. A manual check is necessary to validate the results of the dynamic analysis. The approach abstracts states from execution traces. Only transitions that affect the DOM state are considered. To identify those transitions the code of the application is statically analyzed. The FSM is then used to extract test cases of semantically interacting events that could reveal faults related to the asynchronous nature of AJAX applications. This approach has been compared to existing white-box and black-box techniques [MRT08] and proved to be complementary to these techniques revealing faults that cannot be revealed by other techniques.

An open question for model based approaches is how to cover a model in a feasible way without loss of effectiveness in fault detection. Even for small applications the model can be complex making the construction of test sequences that sufficiently cover it costly or even infeasible when cycles exist in the model.

By definition these models also only represent the application as was intended by the developer of the application: Connections between nodes are derived from links and `forms` that are identified in the application's code. However, because the user of a web application has extended control over the application supported by browser functions,

such as `Back` and `Refresh` as well as the address bar and bookmarks, in theory any two actions can be performed in sequence even if their connection is not represented in the model. Therefore, restricting test sequences to those that can be found in such models would not account for all possible sequences that can occur in practice.

Session Data Based Testing

The sequence of actions that the user performs from entering until leaving a specific web site is known as a user session. Many web applications are configured to log user sessions to help customize the content or analyse usage patterns to enhance the application's design. These sessions capture real user behaviour and usage of the application making them potentially valuable in testing. Using these sessions to generate test suites also addresses the problem of automated test data generation, which in model based testing required manual input from the tester.

Elbaum et al. [EKR03] were the first to propose a web testing approach that uses data captured in user sessions to create test cases automatically. The web server's session logs were analysed and partitioned into single user sessions using IP addresses of requests and session identifiers. The application can then be tested by replaying individual sessions or by mixing part of different sessions. Elbaum et al. evaluated this session based approach by comparing it to existing model based techniques. The evaluation compared session based approaches to two implementations of Ricca and Tonella's model based approach [RT01a]; a simple implementation that uses only one value per field and an implementation that employs input ranges to populate fields in a test case. They also evaluated a hybrid approach that assigns values collected from session data to input fields in the model based test sequences. The evaluation revealed that Ricca and Tonella's approach with input ranges performs the best in terms of coverage and fault detection. However, the session data based technique's results were competitive and complementary to the model based approaches but at a lower cost and effort since the session based approach is fully automated. The evaluation also revealed that mixing different parts of sessions and the hybrid approach did not provide any additional benefit.

A later study [ERKI05] defined new strategies for session based approaches and hybrid approaches. The new session based approach replays sessions but mutates input values automatically by randomly deleting characters from their values. This approach

proved to be the most effective in detecting faults outperforming the model based approach. However, the evaluation was performed on one application and the sessions were collected for the experiment rather than real sessions and were relatively small in numbers.

When an application is configured to record session data, the amount of recorded sessions can grow large very quick. Rerunning all recorded sessions can be inefficient because a lot of redundancy in requests can be expected. For example, a large percentage of users are expected to login to a web application or perform a search. Sampath et al. [SMSP04] designed a testing framework that is based on user session data and attempts to address the problem of test suite size. To automate the test data generation process, the framework updates a test suite with test data generated from user session data throughout the application's lifetime. To maintain the size of the test suite, the framework continuously reduces the test suite using concept analysis. To perform concept analysis, the sessions were considered objects and URLs were treated as properties [SSP04]. Input values and sequence order of the URLs were ignored to simplify the analysis. Common subsequences of URL were subsequently examined to help understand web application behaviour and use-cases.

To evaluate the clustering approach, a later study by the same authors [SSG+05a] examined the relationship between cluster size and overlap in coverage and fault detection within the same cluster as well as between different clusters. The results show that clusters with smaller attribute sets display smaller overlaps in coverage and fault detection within the same cluster. However, the overlap between different clusters is much smaller supporting the hypotheses that each cluster represents a different use-case of the application.

This concept analysis based reduction technique was empirically evaluated [SSG+05b] and compared to random and requirements based reduction techniques. The results show that the concept analysis based technique requires less time, space and effort, since it is fully automated, while still providing high coverage and fault detection capabilities. Further work [SGSP05b] defined a framework for session based testing that uses the clustering method and addresses other challenges in this type of testing. Ignoring application state when replaying sessions [EKR03, ERKI05] resulted in new cases that might have revealed new faults. However, it is desirable to account

for state in session based testing to reflect the original user behaviour and also to guarantee consistent behaviour when the test suites are executed repeatedly. Sprenkle et al. [SGSP05b] addressed this issue by initializing the state then replaying the sessions by the order of their first access of the application. The evaluation compares statement coverage of the original sessions and the reduced test suite both with and without state maintenance. The results show that the reduction in coverage is relatively small for the reduced test suite (1.1%) and even smaller when state is ignored (0.1%).

Finally, Sampath et al. [SBV+08] evaluated and compared different prioritization techniques on test cases derived from session data. The techniques used were based on ordering test cases based on test length, sequence frequency, distinct parameter, distinct parameter pairs, number of parameters and random. The results show that none of the prioritization techniques performs better for all considered applications. However, ordering based on the number of parameters and distinct parameter pairs detects all faults for two of the three applications studied faster than any other technique.

Luo et al. [LPC09] also used clustering to reduce a session-based test suite. Their approach uses a service profile derived either from the system's specifications or by reverse engineering the application to assign sessions to clusters. The assignment is based on the number of common requests and input fields between a session and a service profile. A tool WebMTA generates the application's dependence graph. The sessions with a higher count of dependence relationships in each cluster are included in the reduced test suite until all dependence relationships are represented. Any dependence relationship that is not represented in sessions is added by inserting missing requests or missing input fields into sessions.

The approach was compared to several other reduction techniques including the concept analysis technique proposed by Sampath et al. [SSG+05b] on both a case study application as well as a real system with real faults. The results show that the test suite size was mostly smaller for Luo et al.'s approach and the fault detection ability was clearly higher for both applications than any other technique. However, this is only clearly observable for the approach when the session data is augmented to cover dependence relationships.

These results confirm the initial findings by Elbaum et al. [EKR03, ERKI05] that session based approaches are complementary to white-box approaches. While the

approach is promising, it requires more effort than Sampath et al.'s approach since system specifications and the system dependence graph need to be present.

In session based testing, logs are converted into test cases by partitioning the log into independent single user sessions. This approach does not capture the real application behaviour because in reality requests from different users are intermingled. If the application contains multi-user state dependencies, such as a database, separating sessions would result in different application behaviour to the original behaviour observed when the logs were collected. This could mean additional coverage of code not covered by the original user sessions but could also mean a loss of coverage achieved by the original requests. Sprenkle et al. [SGSP05b] discussed these issues in more detail and proposed new strategies for partitioning the logs into test cases where one session could contain requests from multiple users. The suggested strategies aim to address the issues related to multi-user state dependencies while still having manageable test case and test suite sizes to simplify debugging of faults and maintenance of the test suite.

Web applications change more rapidly than traditional applications. When the structure of the application changes, some of the test cases produced from session data collected before the change might fail. Alshahwan and Harman [AH08][9] proposed a session data repair approach to be used in regression testing. The approach analyses the structure of the modified application and then traverses the repository of session data to identify invalid sessions. The invalid sessions will then be repaired by adding or deleting requests or input parameters using the analysis data of the modified application.

The strength of session based approaches lies in their portrayal of real behaviour of users of the application and their cost effectiveness because no manual effort is needed. However, several drawbacks limit their use in testing web applications: If the application (or part of it) is new or not previously configured to record sessions this approach cannot be used. There is also no guarantee that these test cases provide completeness or high coverage because some parts of the application and use scenarios could possibly never have been encountered by users of the application. As with statistical testing, usage of the application can change unexpectedly and new parts of the application, not tested using these approaches, can suddenly dominate usage patterns.

[9]This paper was based on the author's MSc work which won the best MSc project 2005 prize, King's College London. No part of this MSc work is included in this thesis.

Interface Identification

In web applications, the expected input fields and their types are not explicitly defined. The client and server communicate through global arrays (e.g., `$_GET`, `$_POST`) where the index is the input name and the array element is the value. The values these arrays accept both as indexes and array elements are not restricted to a predefined set. Therefore, identifying input fields and types (i.e. the interface) can be useful in providing guidance to test data generation techniques. Halfond and Orso [HO07] used static analysis to identify interfaces of Java web applications. For each function call that accesses the HTTP request's global arrays to retrieve an input field, the operations performed on the input fields are analysed to get domain information. Operations, such as type casts determine the field's type while comparisons to constants determine the field's possible values. The input parameters that are used along the same path are grouped into interfaces. The discovered interfaces were then evaluated by measuring their effectiveness in generating test cases that achieved higher coverage in comparison to a state-of-the-art spider.

In subsequent work [HO08], the approach was applied to finding parameter mismatches in Java web applications and a tool called Web Application Interface Verification Engine (WAIVE) was developed to implement the approach. In addition to identifying the interface, the tool identifies locations where these interfaces are used then analyses the results to find mismatches. The tool was evaluated on four Java web applications and reported 151 mismatches with only 19 false positives. However, in strongly typed languages, such as Java, the result can be complete while for dynamically typed languages, such as PHP the types of some variables might not be identified using this approach.

The tool was enhanced to use symbolic execution to identify interfaces instead of static analysis. The approach is divided into three steps: symbolic transformation, generating path constraints by executing the application symbolically and finally identifying accepted interfaces. The enhanced tool was used for test data generation [HAO09] by first identifying the interfaces and then using simple test generation techniques to generate test cases. For each input, three types of values were generated: normal, empty and erroneous (e.g., string when integer is expected). The interface discovery results and coverage results where compared to a crawler that uses the same input generation

strategy. The evaluation showed that the new approach discovers more interfaces and achieves higher coverage than the crawler.

Elbaum et al. [ECIR06] also investigated web interfaces. However, their approach was a black-box technique that submits requests to the web server, analyzes the output and infers characteristics about the interface. For a chosen `form`, the approach starts by analyzing the HTML `form` definition to identify input fields and collect possible values (e.g., from drop-down menus, radio buttons). The tester is prompted for values for fields of unbounded type, such as text fields. A set of requests with all possible combinations of input values is generated and a request selector executes a subset of those requests. The output is checked to determine whether it is valid or invalid. An inference engine analyses the result to infer properties of input fields and relationships. The inference engine uses different algorithms to determine whether a variable is mandatory or optional and also which variables need/cannot be present together. It also infers properties about accepted values and ranges. However, the results could be biased since some of the values are provided by the tester. Some aspects of the approach are manual: providing values for text fields and defining what constitutes a valid response. A prototype tool WebAppSleuth was developed to implement the approach. Another limitation of the approach is that it assumes the processed `forms` to be deterministic and stateless. Since many web `forms` are stateful, the current implementation of the approach does not cater for them.

Later the tool was enhanced [IER07] to automatically identify valid and invalid responses: A valid response contains data and an invalid response is empty or contains an error message. The tool was also enhanced to infer a new property: the hierarchal relationship between values where the result for one value should be a subset of the results for another value if all other fields are kept constant. For example, in a product search, the records returned for a minimum price value of 10 should be a subset of the records returned for a lower minimum price value of 5. The evaluation was also extended to include `forms` from six applications including `forms` selected from well-known commercial applications, such as Travelocity and Expedia.

Identifying and understanding the interface of a web application can be very useful in performing many tasks including test data generation. However, to be used in testing, interface identification needs to be used in conjunction with test generation methods.

Bypass Testing

In web applications, input validations can be performed either on the client-side or server-side or both. Client-side validations are triggered by user actions, such as filling an input field or submitting a `form`. Since these validations are performed without interacting with the server, network traffic is reduced and notifications of errors are produced faster to the user. However, the nature of web applications makes it easily possible for users to bypass these validations by submitting requests directly to the server. This can be done, for example, by typing the query string directly into the address bar of the browser or by writing a small program that uses one of the libraries that perform such operations.

Bypassing the interface can be attempted by hackers to exploit vulnerabilities but can also be done by normal users either intentionally or by accident. This can occur, for example, when JavaScript is disabled or when a `form` submission result page is bookmarked and later visited thereby submitting the saved values without executing the client-side validation script. In all these cases, the application could behave unexpectedly exposing sensitive data to the user. Performing input validations on the server-side resolves these issues. Access to the database and application state is only available on the server-side and more powerful languages can be used to implement validations. However, sending requests back and forth to perform input validations increases the pressure on the server and affects performance.

Offutt et al. [OWDH04a, OWDH04b] introduced the idea of bypass testing. The concept of bypass testing is to submit requests directly to the server that violate input validations. The approach analyses input validations in HTML `forms` and JavaScript on the client-side and builds a set of constraints for each input. Input values are then produced that violate these constraints. This type of testing requires no knowledge of the server-side code and preliminary results [OWDH04b] showed that it exposes faults that cannot be discovered without bypassing the interface.

The approach was later applied to CyberChair [OWDH04a], a real application used in many conferences to manage paper submission, detecting a number of interesting faults and vulnerabilities. One of the more interesting vulnerabilities discovered was the ability to submit a paper without successfully logging in to the application. This vulnerability was exposed by making a few modifications to the submission `form`

which can be replicated by any user since the `form` is on the client-side. This experiment showed that the application does not ensure that a user is logged in before accepting a submission. The approach was also enhanced to perform three types of bypass testing: In addition to the previous 'Value Level' bypass testing, parameter and control level bypass testing were performed. Parameter level bypass testing attempts to violate defined relationships between fields while control level bypass testing tries to violate the defined sequence of requests the application accepts.

Bypass testing was applied to an industry case where its effectiveness was confirmed by revealing 63 unique failures in the studied application [OWO08].

Bypass testing raises an important issue in web applications which is the extended control that the user has over the application. Due to the nature of web applications, the user could, as previously mentioned, use the application in a way that was not intended by the application's developer. Faults resulting from this unexpected use should not be dismissed as false positives because they can be replicated by the user through the application's interface. However, the approach can be more effective in revealing faults and vulnerabilities if used in conjunction with more powerful test data generation approaches. Generating test data that violate client-side constraints in bypass testing revealed faults not found by other approaches but combing this process with test cases that achieve higher coverage, can reveal harder to find vulnerabilities in parts of the code that are harder to reach.

Browser Interactions

As previously mentioned, one of the challenges in web testing is that the user has extended control over program execution. In addition to clicking links and submitting `forms`, the user has the ability to use browser functions, such as `Back` or `Forward` to alter the intended execution path. This can result in inconsistencies and unexpected behaviour. Di Lucca and Di Penta [DLDP03] proposed a testing approach that addresses this problem. The approach is proposed to be integrated with any existing testing method. After a testing method is selected and the test suite is built, this approach generates a state chart that takes into account possible browser interactions and generates a new test suite that satisfies a chosen testing criterion. The output is then compared to the output of the original test suite to discover inconsistencies and failures. This approach addresses part of the problem, however it is limited to inserting one in-

stance of `Back` or `Forward` between two requests in the original test sequences. In practice, any number of those actions can occur. Moreover, alterations to the execution path caused by, for example, typing a URL into the address bar of the browser or visiting a bookmark are not modelled.

Licata and Krishnamurthi [LK04] also addressed this issue by using model checking. The model checker describes the application's behaviour as well as possible user alteration of flow. The approach addresses the case where a user clones a page (i.e. opens a link in a new window), returns to the original flow of the program but then submits the cloned page. The motivation of the approach is an error found in Orbitz where when the previous scenario was executed, the item (in this case flight) that was submitted (reserved) was the last item viewed by the user rather than the item on the page where the submission button was clicked. The approach was designed for programs written in PLT scheme that use the send/suspend primitive, which does not apply to more widely used web applications written in PHP or Java. The send/suspend programming procedure assumes that only one possible continuation point can be resumed.

Those two approaches attempt to partially model user's possible changes to the intended execution flow of the application. In practice, those changes are unlimited because in principle any two requests can be executed in sequence. Because of security issues that present a threat in web applications, all execution paths need to be tested. However, that is often infeasible making it necessary to find approaches that prioritize execution sequences that are more likely to expose faults without restricting the method used to build those sequences.

Dynamic Symbolic Execution

Dynamic Symbolic Execution is a combination of concrete and symbolic executions of the application under test. The approach is based on executing the application symbolically with an initial set of inputs (typically an empty set) and building a path constraint. New path constraints are then generated by using subsets of the previous constraint with the negation of one conjunct. A constraint solver is then used to produce new inputs that satisfy these constraints if possible. This approach has been used in web applications to produce test data to cover server-side code or uncover vulnerabilities that could lead to security threats.

Wassermann et al. [WYC+08] used Dynamic Symbolic Execution to generate test data for PHP web applications. The test data was generated to check for possible SQL injection attacks. For scalability, only constraints related to the parts of the program that call database queries were examined. The technique was able to find vulnerabilities that may lead to SQL injections that were not found by static techniques.

Artzi et al. [AKD+08] used a similar approach to generate test data and find bugs in PHP applications. A tool called Apollo was developed to implement the approach and generate inputs. To perform symbolic execution, the code was transformed to simulate user actions, such as button pressing. The transformation is done manually which limits the automation of the approach. This manual effort was eliminated in later work [AKD+10] by adding a state manager and analyzing the output HTML for new user actions. However, using this approach that relies on the output HTML to discover user actions dynamically might miss parts of the application. It also limits sequences of requests to those defined by the developer of the application while in web applications, as discussed previously, the user has extended control over the application.

Figure 2.5 depicts the architecture of the tool. A modified version of the PHP interpreter (Shadow Interpreter) is used to perform the test generation process. the 'shadow' interpreter performs both concrete and symbolic executions simultaneously. The tool also monitors the output for PHP errors and malformed HTML and collects the test cases that caused them. To localize faults, a post-processing step is performed to minimizes the conditions on inputs that cause failures. The tool was evaluated on six PHP applications and compared to random test data generation. The faults found by Apollo included malformed SQL errors that are a result of concatenating input fields into SQL statements without validation leaving the application vulnerable to SQL injection attacks.

Emmi et al. [EMS07] proposed a Dynamic Symbolic Execution approach that, in addition to constructing constraints on inputs, tracks the database state and its constraints. A symbolic map is maintained for the database and the database state is changed for each execution based on the suggestions of the symbolic execution. In addition to the database's effect on coverage, the database query statements could contain conditions that can be considered branches. Since SQL is usually embedded in a host language, such as Java or PHP that implements the application, coverage criteria

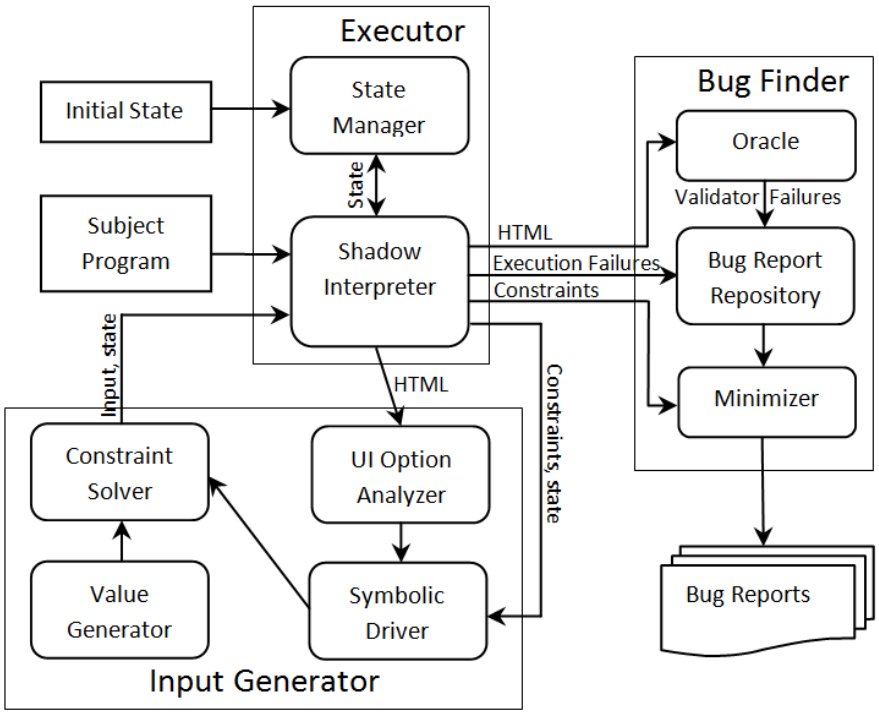


Figure 2.5: Architecture of the Dynamic Symbolic Execution tool Apollo reproduced from the work of Artzi et al. [AKD+10]

of the host language often overlooks these branches. The main limitation of the approach is scalability but also changing the database is done artificially (not through the applications interface) which might introduce false positives.

Web Crawlers

One of the first attempts to analyze and fill `forms` automatically was proposed by Doorenbos et al. [DEW97] for a shopping agent that helps the user locate the best deals for products on the web. Although the scope was limited to searching for specific products provided by the user, the approach used to identify fields and analyse the output has been employed by later studies for the purpose of testing. For instance, for the shopping agent to learn what an error page looks like (e.g., 'product not found' page), a number of `form` submissions are tried first with random strings and the output is analysed. To identify `form` fields that are relevant to the product the agent is investigating, field names and adjacent labels are matched to the product information provided by the user of the agent.

Raghavan and Garcia-Molina [RGM01] developed a crawler called the Hidden Web Exposer (HiWE). The crawler was not specifically developed for testing but for

extracting data hidden behind forms. When the crawler encounters a form, it first analyses the form definition to extract fields, labels and values. The labels are used to obtain more information about the fields in order to generate meaningful values. When filling form fields that are unbounded (e.g., text boxes), HiWE uses a table populated with values from a combination of user provided values, built-in categories, such as time and date, crawler experience and data extracted from well-defined web directories, such as Yahoo. The labels of fields in a form are matched, using approximate string matching, to the labels in the table. This approach makes values reusable in similar fields with slightly different names. To identify error pages, the crawler analyses the HTML structure of a response page to find the most significant part of that page. The hash values of that significant part are then stored for each form on multiple submissions. When a hash value for a particular form appears frequently, the response page is considered to be an error.

Assigning a value to a form field could be a trivial task in case of fields with a limited number of possible values, such as checkboxes or radio buttons. However, the task becomes more complicated when the crawler is faced with a field with infinite possible values, such as a text box. Some fields are unbounded but optional or only needed for extra information, such as a description field or a name field. Any value provided for such a field would result in a successful submission of the form. However, some fields are mandatory and require specific values: validations are performed on these fields and the value provided has to be valid for the submission to succeed. An example of such fields is a user name field or post code.

Liddle et al. [LESY03, LYE02, LYE03] describe a domain independent crawling mechanism that, similar to HiWE, is focused on data retrieval. For a specific form, the aim is to attempt to retrieve all data that lie behind that form. The strategy followed is to first submit the form using default values found in the form definition. A number of subsequent submissions are made and the result compared to the initial submission. If no new data was found, the initial default submission is assumed to have returned all data. Otherwise, other combinations are tried until a user defined stopping criterion is satisfied.

VeriWeb [BFG02] is a tool that combines the benefits of crawlers and capture/reply tools. The tool has the ability to discover and fill forms automatically using user-

defined profiles. VeriWeb executes within a browser environment, similar to capture/reply tools, making it able to execute client-side scripts. The tool explores a web application from an initial state to a given depth recording a trace of actions taken that can be later repeated interactively by the tester. An error reporting mechanism is also provided that, together with the recording capability, is useful for debugging. The tester can specify profiles of input data that should be used as a set (e.g., username and password). The profiles are specified on the application level and not on a `form`-by-`form` basis to be more flexible and limit the manual effort required by the tester. For each field, a value or a regular expression is provided. Aliases for the same field can be specified since a slightly different name might be used in different `form`s. Policies can also be specified to limit the number of possible completions of a certain `form`. For example, a policy to use only one profile per execution path can be defined. Unlike the previously discussed crawlers, VeriWeb is specifically developed for testing. VeriWeb requires considerable manual intervention from the user in defining input profiles (i.e. the input values). As with all crawlers, the test cases generated are large, depending on crawling time, which makes it hard for the tester to check the output and debug errors.

Static Analysis

Minamide [Min05] implemented a string analyzer that approximates the string output of a PHP program. A PHP program together with input specifications are passed to the string analyser. The input specifications provided by the user describe input types and names as well as return values of functions. The analyser first translates the program into a functional intermediate language equivalent to the static single assignment form. The tool extracts a context free grammar from this and uses it to approximate the program's output. The approximated output can then be used to validate HTML or check for cross-site scripting vulnerabilities. The approach was tested on six PHP applications and was able to detect both HTML faults and cross-site scripting vulnerabilities. Web applications are dynamic and PHP specifically has many mechanisms that can construct code dynamically. Therefore, static analysis fails to provide a complete analysis of code and later work that relies on dynamic methods by Artzi et al. [AKD+08, AKD+10] has been shown to outperform Minamide's results.

Formal Methods

Jia and Liu [JL02] apply formal specifications to web applications. They define a specification language that caters for special characteristics of web applications. They developed a tool WebTest that accepts formal specifications of the test suite as well as the expected response of an application as input, executes the tests and outputs a test report. Security and performance testing test suites can also be described by the formal specifications and tested using WebTest. This approach is useful when such specifications are already present; otherwise the benefit of the approach is limited to automating test execution.

Testing Cookies

Web applications use cookies as one of the ways to overcome the stateless nature of HTTP requests. Cookies are small files that reside on the client's file system and store information that the server can use to identify the client. For example, if the client uses a feature to be kept logged-in to an application on subsequent visits from the same machine, the server could store a cookie that makes it possible to identify this user's choice in the future. Tappenden and Miller [TM08] defined a strategy for testing cookies and their effect on the application's behaviour. A grammar for cookies was defined and several testing strategies were explored that are specific to the nature of cookies. Their approach is independent of structural testing of the application and no evaluation of the proposed approach was presented.

2.4 Search Based Software Engineering

Search based software engineering is an approach that formulates software engineering problems into optimization problems [HJ01]. The approach uses metaheuristic search algorithms to find near optimum solutions for problems where no optimal solution can be found. The approach has been applied to many areas of software engineering throughout the development lifecycle including cost estimation, requirements engineering, test data generation and test data optimization [Har07].

To apply search based techniques to a problem a number of decisions need to be made. First, the possible solutions need to be ordered in a way that makes similar solutions adjacent. For example, if the solutions are integer numbers, the neighbours of the number 3 are 2 and 4. This ordering process is not trivial for more complex

data types, such as records or strings. Second, an objective function or fitness function needs to be defined and used to compare solutions. Finally, operators to alter unsuccessful solutions need to be selected in a way that directs the search to a 'better' solution. For integers, for example, such operators could be addition and subtraction. Once the problem is formulated in this way, a search based technique can be applied to solve the problem. Many techniques are currently used and many new techniques are emerging. Hill Climbing [Kor90], genetic algorithms [MMS01], evolutionary strategies [AC06, AC08], tabu search [Glo90] and simulated annealing [TCMM98, TCM98] are examples of such algorithms.

The remainder of this section reviews existing concepts used in search based test data generation. These concepts are employed and enhanced in part of this thesis in the application of search based test data generation to web applications in Chapter 3.

2.4.1 Search Based Test Data Generation

Miller and Spooner [MS76] were the first to suggest a dynamic test data generation approach that uses search in 1976. The paper suggests an approach for generating floating point test data while fixing integer parameters. A path is picked for generating test data and all constraints that satisfy that path are used to compute the fitness of the input data. Constraints are computed by transforming them in a form that would result in a positive value if the constraint is satisfied. For example, a constraint that takes the `true` branch of the test `if` $(X \neq Y)$ has a constraint $c > 0$ where $c = abs(X - Y)$ or $c = (X - Y)^2$. The fitness is only positive if all constraints are positive (i.e. satisfied). The approach begins with a random set of inputs and the values are then adjusted until the fitness is positive and thereby the inputs that satisfy the chosen path are found.

In 1990 Korel [Kor90] developed Miller and Spooner's approach to simplify and enhance the search process. Predicates of the form X `op` Y where X and Y are expressions and *op* is a relational operator are transformed to F `rel` 0 where F and `rel` are described in Table 2.2. A path is then selected for which the approach wants to generate test data to cover. An initial random input vector is generated and executed on the program under test. If this input vector causes the target path to be covered, the input vector is recorded. If the path was not covered, search based techniques are used to try to cover the branch that caused the execution to divert from the target path.

Branch Predicate	Branch Function F	rel
$X > Y$	$Y - X$	$<$
$X \geq Y$	$Y - X$	$\leq$
$X < Y$	$X - Y$	$<$
$X \leq Y$	$X - Y$	$\leq$
$X = Y$	$abs(X - Y)$	$=$
$X \neq Y$	$abs(X - Y)$	$\leq$

Table 2.2: Korel's transformations of branch predicates to calculate fitness

Once a solution has been found, the process is repeated for each divergence point until the target branch is covered or the approach fails to produce a solution. This differs from Miller and Spooner's approach in that fitness is calculated for branches rather than all constraints in the path. Korel also introduced the Alternating Variable Method (AVM), which is a variation of Hill Climbing into the search process. This method makes changes to one input variable while fixing all other variables. Branch distance is used to measure how close an input comes to covering the traversal of a desired branch. If the changes to a variable affect branch distance, AVM applies a larger change in the same direction at the next iteration. This 'acceleration' could cause the approach to 'over shoot' the nearest local optimum. In this case, AVM restarts its search at the previous best solution so far. The approach will then cycle through variables, repeating the same process, until the branch is covered or no further improvement is possible. Dynamic dataflow analysis was also used to determine the order in which input variables are picked for the Alternating Variable Method. Variables are assigned weights based on their influence on the target branch to improve the performance of the approach.

Ferguson and Korel [FK96] developed Korel's work by introducing the chaining approach. In some cases reaching the target node is dependent on the execution of a node that defines the value of a variable used in the target node. This dependency is not captured by the control flow graph and therefore using the control flow graph is not sufficient to guide the search. The chaining approach analyses data dependencies and uses this analysis to generate inputs that execute the affecting definition statement and thereby improve the chances of covering the target node.

Gallagher and Narasimhan [GN97] used search based techniques to generate test data for Ada83 programs. The approach takes into account *AND, OR, NOT* and *XOR* operators while previous work assumes predicates are free from these operators. The fitness function value for a predicate containing an *AND* operator is the sum of fitness values of its two operands while for the *OR* operator the fitness is the minimum. The work also supports exceptions and procedures which Ferguson and Korel's work does not support.

Tracey et al. [TCMM98, TCM98] used simulated annealing global optimization to test functional and non-functional properties of Ada programs. The fitness function was changed to evaluate to zero when a correct solution is found so the aim was to minimize the fitness function. Table 2.3 describes how fitness is calculated. A value K is added when a condition evaluates to false to avoid the need to have different relation operators for different predicate types. When generating test data to cover a certain branch, the fitness is calculated globally by evaluating how close the executed path was to the path that leads to the target branch. This work also incorporates boundary value testing into the testing process by assigning optimum fitness to values that satisfy the predicate of the target branch but are on the boundaries of the range of optimal solutions and slightly penalizing all other correct solutions. Assertion and exception testing is also performed by transforming assertions and exceptions to branch predicates that can be covered using the same search based structural testing techniques. An application of the approach to non-functional testing was also suggested where the same algorithm was used but the fitness function was the execution time of a given test case.

Alba and Chicano [AC06, AC08] used Evolutionary Strategies (ES) to generate test data. Evolutionary Strategies simulate evolution in nature to generate test data. An initial population of test cases is created randomly or by a seeding algorithm and then executed and evaluated on the system under test. In the next iteration, the result of the evaluation is used to evolve the population generating a new population, or offspring, from the fittest parents. The input vector is represented as real and integer numbers together with additional self adaptive variables that are used in mutation. The approach aims to maximize condition-decision coverage; for every condition in the program, test data is generated for the condition to evaluate to both `true` and `false`. While targeting a condition, fitness of other conditions is monitored to capture test cases that

Element	Value
Boolean	if TRUE then 0 else k
$a = b$	if $abs(a - b) = 0$ then 0
	else $abs(a - b) + k$
$a \neq b$	if $abs(a - b) \neq 0$ then 0
	else k
$a < b$	if $a - b < 0$ then 0
	else $(a - b) + k$
$a \leq b$	if $a - b \leq 0$ then 0
	else $(a - b) + k$
$a > b$	if $b - a < 0$ then 0
	else $(b - a) + k$
$a \geq b$	if $b - a < 0$ then 0
	else $(b - a) + k$
$a \vee b$	min(cost(a), cost(b))
$a \wedge b$	cost(a)+ cost(b)
$\neg a$	Negation is moved inwards and propagated over a

Table 2.3: Tracy et al. [TCMM98] Fitness Function for local distance.

satisfy other conditions by accident and thereby increase the efficiency of the approach. Alba and Chicano compared evolutionary strategies to Genetic Algorithms and found that ES perform better for some of the programs studied. As well as normal branch coverage, 'Correct Coverage' was introduced which excludes infeasible branches from the normal branch coverage measures.

Michael et al. [MMS01] used a minimizing fitness function similar to the one used by Tracey et al. but generated test data for C and C++ programs using Genetic Algorithms (GA). A new approach was proposed for selecting the next branch the GA targets; branches are only targeted if they have already been reached. The next branch is selected from nodes where either the `true` or `false` branch has already been covered and the GA will attempt to cover the other branch. The test inputs that cause a branch to be reached are used as a starting point when that branch is targeted later rather than using random test inputs. Like Alba and Chicano, test cases that accidentally cover branches, other than the target branch, are captured. An interesting observation from the evaluation is that accidental coverage of test criteria was more common than deliberate coverage.

Michael et al. also compare different search based algorithms including GA to a random test data generation approach on both simple programs and real world more complex programs. The results of this comparison indicated that the difference in performance between random and GA becomes more pronounced when programs have higher complexity.

In many cases, when trying to cover a target branch, several paths in the program could lead to the target branch but some of those paths might make covering the target branch infeasible. In a standard search based approach this might lead to failing to cover the branch since the search process was not designed to handle those cases. McMinn et al. [MHBT06] proposed a program transformation that factors all paths that lead to branches giving the search process more guidance in finding the correct path that leads to covering a branch. They applied this approach to evolutionary testing maintaining a separate population per path. The evaluation showed that this approach is effective for branches that have this property.

Lakhotia et al. [LHM07] suggested that, in practice, a test data generation process might seek to satisfy multiple objectives simultaneously. Therefore, a multi-objective

test data generation technique is more realistic. The proposed test data generation process tries to satisfy a secondary objective as well as maximizing structural coverage. In a case study presented in the paper to demonstrate the approach, the sub-objective was selected to be maximizing allocated memory.

Generating test data to cover string predicates requires the definition of custom fitness functions and mutation operators. Alshraideh and Bottaci [AB06] were the first to propose a method for search based test data generation for string predicates. Three fitness functions were compared: Hamming distance, character distance and ordinal edit distance (Levenshtein distance). In the evaluation, the Levenshtein distance performed better than the other two fitness functions. The paper also proposes the use of constants gathered from the source code to initialize and mutate string inputs. This proposed constant seeding approach considerably improved performance (up to 5 times). The approach was evaluated on a number of small JScript functions (maximum LOC 62). The improved performance gained by seeding constants collected from the source code indicates a potential source of enhancement of search based algorithms by considering other seeding sources.

Yoo and Harman [YH10] proposed a new source of seeding in regenerating test data from existing test data using search based techniques. The approach suggests that when test cases already exist, either from manual testing or previous releases, test data generation tools perform better by using this previous test data as a starting point instead of randomly generated test data. The evaluation confirmed that using this approach, search based generation tools achieve competitive coverage and mutation scores compared to standard approaches that start from randomly generated test data while being up to two orders of magnitude more efficient.

Zhao et al. [ZLM10] generated string test data for domain testing. Domain testing is aimed at finding input values that are near the boundaries of the domain since these are expected to reveal faults. For example, to generate inputs for domain testing for a program that contains a condition of type `if` $(x \leq 5)$ (where x is an integer) two values need to be generated: one value on the border (5) and one value just outside the border (6). The fitness function used was the string ordinal distance where strings are mapped to a non-negative number using weighed summation of the string characters ASCII code.

McMinn [McM04] surveyed some of the search based techniques used in test data generation for structural, grey-box and non-functional testing. The survey focuses on genetic algorithms, Hill Climbing and simulated annealing and also discusses future direction for work in this area. Ali et al. [ABHPW10] recently published a systematic review of the methods used to empirically evaluate test data generation using search based techniques.

Concolic testing or Dynamic Symbolic Execution is one of the main test data generation techniques together with search based techniques. In concolic testing, the program under test is executed with some random or empty values and the constraint for the path that was executed is built. The approach tries to then cover more paths by negating the last conjunct of the path constraint. The process continues until no more paths can be covered. Lakhotia et al. [LMH09, LMH10] performed an empirical study that compares the performance of concolic testing and search based test data generation. The results of the study showed that the two approaches are complementary and each covers a subset of distinct branches. The study also showed that using these two techniques out of the box for large real world systems achieves moderate branch coverage ($< 50\%$).

Hill Climbing

Hill Climbing is a local search algorithm often used in SBSE and found to be effective for testing [HM07]. A random solution is first chosen from the search space and evaluated. The neighbouring solutions of that random solution are then evaluated to find a better solution. If a better solution exists, that solution is selected to replace the previous solution. The process is repeated until a solution is found for which no further improvements can be made.

The approach has the advantage of being simple and fast. However, the approach's success depends on the randomly chosen starting solution as some initial values can cause the search to be trapped in a local optimum. An improvement on this approach to overcome this problem is to introduce 'restarts' where when a solution cannot be found the process restarts with a new random solution. The previously discussed Alternating Variable Method proposed by Korel [Kor90] is a variation of the Hill Climbing algorithm.

Hill Climbing is not as popular as Evolutionary Algorithms in search based test data generation [ABHPW10]. However, Harman and McMinn [HM07, HM10] investigated situations where the use of Evolutionary Algorithms is beneficial compared to Hill Climbing. The results showed that in many cases the sophisticated, but often more expensive, EA algorithms are not required and a simple algorithm, such as Hill Climbing can be just as effective (and in some cases more effective). The predicates that benefit from using EA need to exhibit certain properties. When these properties are absent, Hill Climbing can outperform EA.

Fitness Function

The fitness function is the primary elements in a search based algorithm as it provides a way of evaluating solutions and thereby guiding the search. Two types of fitness functions have been used in the literature for test data generation that aims to maximize structural coverage. The first type is coverage oriented while the second type is control oriented. In coverage based fitness [Rop97, Wat95], the fitness of a test case is evaluated based on how much coverage it achieved. The problem with this approach is that it provides little guidance to the search algorithm to cover new branches. The design of the fitness function will also often drive the generation process to test cases that have longer execution paths that are easier to find [McM04]. Control based fitness functions can be either based on branch distance [Kor90, JSE96] or control flow oriented [PHP99] or a combination of both [WBP02].

Branch distance based fitness functions [Kor90, JSE96] calculate the distance of the branch that diverted execution from a path to the target branch. This approach divides the coverage of a target branch to several sub targets: These sub targets are the branching points along the execution path that leads to the target branch. This approach forces the search process down a specific execution path and therefore might cause it to be trapped in a local optimum.

Pargas et al. [PHP99] used a control flow oriented fitness function when using genetic algorithms to generate test data. The control flow based fitness function calculates fitness by comparing the predicates in common between what a test case covers and the predicates in the path that leads to covering the target branch. The higher the number of common predicates the higher the fitness. However, this fitness function fails to capture another difference that affects fitness. Two test cases that cover the same number

of common predicates will have the same fitness even if one is closer to covering an additional predicate.

A fitness function that combines both control flow and branch distance based fitness can overcome the issues related to using each type of fitness individually. Tracy [Tra00] introduced such a combined fitness function. The fitness for a certain test case is calculated as [McM04]: $Executed/Dependent \times Branch\ distance$ Where $Dependent$ is the number of branches that need to be covered to reach the target branch and $Executed$ is the number of those branches that were covered by the test case. $Branch\ distance$ is the local distance of the branch that caused the execution to divert from the path that leads to the target branch. Although this fitness function avoids the issues faced in previous fitness functions, it might favour test cases that executed longer paths.

Wegener et al. [WBP02] propose a similar fitness function that overcomes this problem by introducing the approximation or approach level. The approximation level counts the number of branches that need to be covered between the branch that caused the execution to divert and the target branch. This is added to a normalized local branch distance of the branch that diverted execution from the path leading to the target branch.

In the approach proposed by Michael et al. [MMS01], the processing of a target is delayed until test generation for other targets had caused the target to be reached. This approach is not useful when the goal of the test generation process is to cover a few specific targets. However, when the goal of the test generation process is to maximize coverage of the system under test, this approach eliminates the overhead of analyzing the control flow of the program to calculate the approach level.

2.4.2 Search Based Techniques in Web Applications

The majority of web related search based research targeted web services which are different in their structure and testing challenges [BHH12] to web applications.

Tsai et al. [TZPC05] used search based algorithms to cluster the output of web services that perform the same function to aid in a voting mechanism that decided which web services return a faulty output. Simulated annealing was used to automatically group the output from different web service into clusters to identify which services are valid. This approach can be useful when a web service used by the application becomes

unavailable and an alternative service needs to be found. The approach suggests that if the majority of similar web services return a certain output, it is more likely that the output is correct. Clustering is used because the output of web services may be slightly different but still valid but also because using clustering makes the approach application specific.

Blanco et al. [BGFT09] used a scatter search algorithm to generate test sequences to cover transitions in the state machine of a BPEL business process. A BPEL business process is a business process that uses BPEL specifications to describe the possible combinations of web services that can be called to carry out its functionality.

Gu and Ge [GG09] used genetic algorithms for performance testing of web services. They generated test cases that produce the best and worst execution times of a composition of web services.

Although published work on search based algorithms is dominated by testing, the application to web application testing is very limited and test data generation has not been applied yet to web applications. Marchetto and Tonella [MT09] used search based techniques to test AJAX applications. AJAX (Asynchronous JavaScript And XML) is a technology that allows the client and server to communicate asynchronously and dynamically manipulate the DOM (Document Object Model). The DOM is the collection of objects that are presented to the user, such as tables, forms and buttons. This asynchronous communication and manipulation of the DOM introduces new types of faults not found in traditional web applications. For example, a wrong assumption about the DOM by the application can lead to faults.

The approach first uses execution traces to build a finite state machine of the application and identify semantically interacting events. Two events are semantically interacting if swapping their execution order produces a different state. A Hill Climbing search based algorithm is then used to generate test sequences that maximize diversity of the test suite. Several fitness functions that measure diversity were used: frequency of event calls, frequency of pairs of events and coverage of the FSM. The approach produces test suites that are smaller than exhaustive testing with a small degradation in fault finding ability. This work was later extended [MT11] by using simulated annealing to avoid Hill Climbing's tendency to be trapped in a local optimum. A new diversity fitness function based on sequence length was also introduced.

Türpe [Tür11] proposed a road-map for security testing of web applications using search based techniques. They suggest that search based techniques can be used to find diverse but interesting starting test cases for web application security testing. New test cases can then be generated that are similar to the starting test cases that exposed a security vulnerability but are better at exposing those vulnerabilities.

2.5 Test Adequacy Criteria

Test adequacy criteria are used to decide when a system under test has been sufficiently tested. Test selection criteria are used to select a subset of test cases that are expected to be more effective in revealing faults. A test adequacy/selection criterion can be specification based: Test cases are generated to cover all specifications of the system. However, specifications might not always be available or complete. Therefore, program based adequacy/selection criteria have been proposed to guide automated test data generation. The main two categories of program based criteria are control flow based and dataflow based criteria. For graphical user interface (GUI) testing, special criteria that are based on events and event interactions were developed [MSP01] to cater for the special nature of GUI systems.

There has been much previous work [ABLN06, HFGO94, FW93, NA09] that investigated the effectiveness of those program based criteria in finding faults. Specifically, whether those criteria independently influence a test suite's effectiveness in finding faults or if the increased effectiveness is just a side effect of larger test suite sizes that are required to satisfy those criteria. Results confirmed that a test suite's coverage level of program based criteria influences effectiveness. However, results showed that those criteria are not the only factor that affects fault detection. This suggests that new criteria should be introduced and evaluated that complement program based criteria. Andrews et al. [ABLN06] also reported that cost-effectiveness for the different program based criteria is the same. That is, while more demanding criteria are more effective in finding faults, satisfying them often requires larger test suites.

2.5.1 Control Flow Based Testing Criteria

Statement coverage is the simplest and least demanding control flow based test adequacy/selection criterion. A test suite satisfies statement coverage when every state-

ment in the code is covered by at least one test case in the suite. To satisfy branch coverage, every branch in the system under test has to be covered. Path coverage is the strongest and most demanding criterion and requires that every path in the program is covered. The number of paths in a program can be very large and in some cases infinite in case the program contains loops. Therefore, less demanding variations of this criterion have been proposed. For example, the tester can attempt to cover all independent paths or all simple paths. Zhu [Zhu95] provides a formal definition and assessment of these control flow based test adequacy criteria.

When using test adequacy criteria, the test suite is required to have at least one test case that covers each element described in the criterion. For example, for statement coverage every statement in the system under test should be covered by at least one test case in the test suite. However, it has been suggested in the literature that different faults have different levels of probability of propagating to the output and causing a failure. Therefore, effectiveness of test suites can be improved by estimating how likely a fault in a program entity (a statement for example) would propagate to the output. This estimation can be used to compute a recommended number of times an entity should be executed by different test cases before gaining confidence that it has been tested adequately. Chen et al. [CUR+02] empirically investigated the benefits of this method and concluded that although overall effectiveness of test suites was statistically significantly improved, the observed improvement was limited. However, in critical systems, the approach might still be cost-effective since higher confidence in the quality of the SUT is crucial.

2.5.2 Dataflow Testing Criteria

The use of dataflow analysis in test case selection was first proposed by Rapps and Weyuker [RW82, RW85] in 1982. These dataflow based criteria examine definitions and uses of variables when generating test cases. The authors provided a formal definition of dataflow testing by defining the dataflow graph and proposing several dataflow testing criteria. The coverage criteria introduced for the dataflow graph range from 'weaker' criterion (All-nodes) that would require the smallest test suite to be satisfied to the 'strongest' criterion (All-paths) that would require larger test suite to be satisfied but can be infeasible in the presence of loops.

Laski and Korel [LK83] also examined dataflow testing and suggested two testing strategies: The first strategy is to generate test cases to cover all definition-use (DU) pairs for each variable individually, while the second strategy applies the same principle for vectors of variables.

Harrold and Rothermel [HR94] adapted dataflow testing techniques to classes of Object Oriented systems. Three levels of DU testing are proposed: for a single method, for a method and all methods it calls and finally for possible interactions of public methods within a class. Since these interactions are infinite, only a subset of those method sequence calls is considered. Martena et al. [MOP02] extended Harrold and Rothermel's approach by also considering DU pairs of objects. Symbolic execution was used to identify pre and post conditions for paths on method calls.

Hutchins et al. [HFGO94] compared control flow and dataflow based criteria in their effectiveness in fault detection. Two forms of coverage criteria where considered: all-DU and branch coverage. The results showed that achieving 100% coverage for either criterion does not guarantee finding all faults in the SUT. The study also concluded that the two coverage measures are complementary in their fault detection effectiveness. Although the results are interesting, the empirical design contained many sources of possible bias. For example, the faults considered were seeded manually. To construct the test suites used in the study, the mechanism is engineered rather than random introducing bias into the process.

Frankl and Weiss [FW93] compared All-uses and branch coverage (All-edges) and found that for five out of nine subjects considered in the study, All-uses was significantly more effective than All-edges. Since satisfying All-uses requires more test cases than All-edges, effectiveness was compared for suites with similar sizes to eliminate the effect of size. The results showed that in four out of the nine subjects, All-uses still performed significantly better than All-edges.

2.5.3 Web Specific Testing Criteria

Test adequacy criteria for web applications initially focused on coverage of a model of the application constructed by crawling the application. In Yang et al.'s [YHWC99, YHW+02] proposed web testing architecture the definition of traditional structural test adequacy criteria was adapted to the web application's model. For example, statement

or branch coverage were used but the definition of a branch or a statement was modified: Web pages are considered nodes or (statements) and links between pages are considered branches.

These new criteria were adopted by subsequent research, such as Ricca and Tonella's model based approach [RT01a]. Ricca and Tonella's also extended these criteria to redefine dataflow criteria for the web application's model. For example, all-uses criterion is defined as covering every navigation path in the model from every definition of a variable to every use of that variable.

Although testing different paths of the navigational model of the application is important, the use of traditional coverage criteria of the applications code is still needed especially with the rapidly increasing complexity of web application code. Tonella and Ricca [TR04a] proposed a 2-layer testing approach where code level coverage criteria are combined with the high level model coverage criteria. Instrumenting the server-side code to measure coverage is a straight-forward process that resembles coverage instrumentation in traditional applications. However, for HTML and JavaScript the task is more challenging since the code executes on the client-side and therefore coverage statistics cannot be easily conveyed to the server. Therefore, Ricca and Tonella suggested methods to overcome this issue: To determine coverage of stand-alone HTML, a special input field is populated with the HTML file name and passed to the server with each request. For JavaScript, the task is even more challenging since no request is submitted and no inputs can be set. The proposed solution is to record coverage data in cookies and then check these cookies on the server-side to update coverage information. Traditional structural coverage criteria of server-side code have been subsequently used to measure and compare different testing approaches [HO07, AKD+10, EKR03].

Dynamic Coverage criteria based on the way requests are formed was proposed by Sampath et al. [SGSP05a]. The criteria are used for test suite reduction and defined in terms of coverage of distinct URLs, URLs and input names, URLs and input-value pairs and also sequences of two or more of each. For example, for the URL and input names criterion, the original test suite is analysed to extract all requests with distinct URLs that use a distinct set of inputs. A reduced test suite can then be generated that contains one request that matches each identified URL/input set ignoring input values. These criteria can be effective for test suite reduction especially when the test suite

that requires reduction is generated from session data (which is the approach used by the authors of this reduction technique). That is, when significant redundancy in input values is expected, however in a structural based generation techniques the test suite is not expected to contain a high volume of such redundancy.

Bellettini et al. [BMT05] used coverage metrics to decide when to terminate a testing session based on user defined thresholds. A tool called TestUML was created that combines a number of techniques to semi-automatically test a Web application. The techniques TestUML uses include model based testing and session analysis to automatically generate user inputs. The tester is required to review and amend input values as well as define the oracle. The tool then executed the test cases on the application until the stopping criteria are met. The stopping criteria are a mix of metrics, such as number of pages, classes or paths covered, that are calculated while executing the tests and used to stop the testing process when the tester specified threshold is met.

Liu et al. [LKHH00a, LKHH00b] extended dataflow testing to web applications. In addition to traditional dataflow modelling of the server code, variables embedded in HTML and XML and transmitted between pages are taken into account. Input variables in HTML are considered as global variables. The control flow graphs and proposed testing levels are amended to take into account indirect data transmission calls between different client-side and server-side objects in the application. Overall, the proposed approach is a straight-forward application of traditional dataflow testing to web applications. The only difference is the analysis of variables embedded in HTML and XML. This seems to be done to include input variables in the dataflow analysis as those inputs are defined in the HTML code and not the server-side code. However, their approach does not include in the analysis the state of the application, represented by session variables and the database. These state variables are the only part of the application that propagates over several requests since HTTP requests are otherwise stateless.

2.6 Utilizing Output in Testing

Dynamic test data generation techniques, such as search based techniques, use feedback from the execution of test cases to guide the generation process. An intermediate output, which is the fitness of a branch, is used to guide the generation of subsequent test cases. The same applies to Dynamic Symbolic Execution [SMA05, GKS05, AKD+08],

where the constraints collected from the execution path are used to generate the next inputs. In these approaches, part of the output is used to guide the test generation process rather than directly used to generate test cases.

In feedback directed random testing [PLEB07, PLB08] the feedback from executing test sequences, such as exceptions and violations is used to exclude sequences that cause such errors from the generation process. This improves the random generation process in focusing on sequences that are more likely to produce new system behaviour. However, the use of output is limited to excluding failing test sequences from being used in generating new sequences. The approach was implemented for Java Object-Oriented applications in a tool RANDOOP [PE07].

In Graphical User Interface (GUI) testing, Yuan and Memon [YM10, YM07] introduced an approach that uses feedback about the state obtained from executing an initial set of test cases to generate new test sequences that are effective in finding new faults. The feedback on how the initial test cases affected the state of elements in the GUI is used to identify interactions between events. These interactions are used to generate new paths (test sequences) that explore different possible interactions. Two events are considered to be interacting events when their execution as a sequence alters their behaviour.

2.6.1 Utilizing Output in Web Testing

In web applications, the output has been used to understand the application's interface, automate the oracle and help crawlers in automatically filling `forms`. Elbaum et al. [ECIR06] in their interface identification approach examined the output to infer properties about the inputs of the application.

A web application's output can contain elements that are non-deterministic, such as time-sensitive data that pose a challenge when automating the oracle for regression testing. Sampath et al. [SBV+08] proposed an approach to overcome this problem. The original test suite is executed on a working version of the application. The output is saved and used to automatically identify failures in subsequent runs. As well as an oracle that flags any difference in output as a fault, other comparison algorithms are used that compare only HTML tag difference, text difference without HTML tags and filenames of the output. A manual inspection of the reported errors showed that the

oracles that only check HTML tags and filenames reported no false positives but the filename algorithm only captured a small percentage of errors.

Jiang et al. [JHHF08] proposed an approach that automatically identifies problems in a web stress testing environment. To accomplish this, the approach studies the sequences of events in the application's log to discover the dominant behaviour and flag anomalies. When a sequence of events diverts from the norm, the sequence is flagged for further examination. The approach uncovered new errors and reduced the effort needed to examine error logs by flagging only a small percentage of the log. The approach does not generate test data but examines log files for unusual behaviour to flag. Some of the cases that were flagged, when examined manually, were discovered to be faults in the application's code. This indicates that this approach could be potentially useful in automatically identifying faults.

Chapter 3

Search Based Test Data Generation

3.1 Introduction

This chapter uses the client-side output and intermediate output of a web application to enhance search based test data generation. The client-side output of a web application contains many sources of input values that have not previously been utilized in a sophisticated test generation technique. The intermediate outputs used to enhance test generation in this chapter are collected dynamically during execution from run-time values of operands in predicates. Both values mined from the client-side output and values collected dynamically at run-time are used for seeding in the search based algorithm presented in this chapter.

Search based testing has been used widely as a way to automate test data generation for traditional, stand-alone applications, thereby making testing less reliant on slow laborious processes. Search based test data generation has also proved to be effective and complementary to other techniques [LMH09, McM04]. However, of 399 research papers on SBST,[1] only one [MT09] mentions web application testing issues and none applies search based test data generation to automate web application testing.

Popular web development languages, such as PHP and Python have characteristics that pose a challenge when applying search based techniques, such as dynamic typing and identifying the input vector. Moreover, the unique and rich nature of a web application's output can be exploited to aid the test generation process and potentially improve effectiveness and efficiency. This was the motivation for this chapter: To seek to de-

[1] Source: SBSE Repository at
`http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/`

velop a search based approach to automated web application testing that overcomes challenges and takes advantage of opportunities that web applications offer.

This chapter introduces an automated search based algorithm and applies it to six web applications. It also introduces enhancements that seed the search process with constants collected statically and values collected dynamically and mined from the web pages produced by the application.
The primary contributions of this chapter are as follows:

1. The introduction of the first automated search based approach to web application testing, and a tool SWAT (Search based Web Application Tester) that implements the approach.

2. The introduction of the use of Dynamically Mined Value (**DMV**) seeding into the search process. The empirical study presented in Section 3.4 shows that this approach statistically significantly increases coverage in all applications studied and also significantly reduces effort in all but one. These findings may prove useful for other SBST paradigms.

3. The results of an empirical study of effectiveness and efficiency of the proposed algorithms in terms of branch coverage of server-side code, fitness evaluations, execution times and fault finding ability.

The rest of this chapter is organized as follows: Section 3.2 introduces the proposed approach, whilst Section 3.3 describes the implementation of the approach. Section 3.4 presents the evaluation together with a discussion of the results. Section 3.5 presents related work and Section 3.6 concludes.

3.2 Approach

A variety of scripting languages can be used to implement web applications including PHP, Perl, Java, ASP and JSP. This thesis focuses on PHP, one of the most popular web scripting languages in current use [TIO12]. This chapter focuses on PHP in order to provide a concrete web application testing tool to implement and evaluate the proposed approach. However, many aspects of this approach may also apply to other web application languages.

The approach aims to produce a test suite that maximizes branch coverage of the application under test. The approach starts with a static analysis phase that collects static information to aid the subsequent search based phase. The search based phase uses an algorithm that is derived from Korel's Alternating Variable Method (AVM) but which additionally incorporates constant seeding and Dynamically Mined Values (**DMV**) seeding from the execution and web pages constructed by the application as it executes.

The rest of this section describes the proposed approach in more detail. Section 3.2.1 discusses issues in applying search based techniques to web applications and the solutions that this chapter adopts. Section 3.2.2 describes the fitness functions used, while Section 3.2.3 introduces the proposed test data generation algorithms.

3.2.1 Issues and Solutions in Web Application Testing

Static and dynamic analysis phases are used to address the issues raised by web application testing and which are either absent or less pernicious in the traditional Search Based Software Testing paradigm.

Issue: Interface Determination

Description: In various web scripting languages, such as PHP, ASP and JSP, the interface is not explicitly specified. There is no 'program header' that specifies how many inputs a program expects nor what their types are. A number of global arrays (e.g., `GET`, `POST`, `REQUEST`) are usually set on the client-side before a request is submitted. These global arrays use the input name as an array index and the input's value as the corresponding array element. These arrays can be accessed by the server-side code at any point in the program.

Solution: In order to determine the 'input interface' automatically, static analysis is performed on the source code to determine the required inputs. It collects each call to the global arrays (e.g., `GET`, `POST`, `REQUEST`) and then extracts the names of the inputs and the associated submit method. The static analysis also notes the location where these inputs are accessed. For every branch that the approach seeks to cover, all input variables that are accessed before that branch are selected to form the input interface. To determine input types, static analysis is performed that determines the type of inputs based on the type of constants to which they are compared or from which they

are assigned. This approach is similar to that of Halfond et al. [HAO09]. However, for PHP applications the analysis does not, as yet, infer types for all inputs and needs to be augmented manually.

Issue: Dynamic Typing

Description: Web development languages, such as PHP, Python and Ruby are dynamically typed. All variables are initially treated as strings. If used in an arithmetic expression, they are treated as numeric at that operation. However, the same input can be treated as numeric in one expression and as a string in a different expression within the same script. This makes it hard to decide the type of variables involved in a predicate, posing a problem when deciding which fitness function to use.

Solution: To solve this problem, types of variables are checked dynamically at run-time using built-in PHP functions and then directed to the appropriate fitness function.

Issue: User Simulation

Description: In dynamic web applications, the user's interactions with the application's dynamic content need to be simulated to test the application as a whole. Web applications usually have a top level entry page that the user accesses first. User choices on the entry page are passed to the server-side code for processing. A client-side page is then generated and displayed to the user. Some applications have other top level pages that can be accessed only through these client-side pages. Identifying these top level pages raises issues when trying to generate test data automatically for an application as a whole.

Solution: The approach's static analysis identifies top level pages that expose new parts of the application accessible only through client pages during the static analysis phase. A file that is not included by any other file is treated as a top level file. The test data generation process is performed for each top level file.

Issue: Dynamic Includes

Description: PHP supports dynamic includes, where the name of the included file is computed at run-time. An example of this is when the user's choice determines the natural language to be used in the text of the application.

Solution: To deal with dynamic file includes, an approach similar to the one proposed by Wassermann and Su [WS07] is used; for include statements that contain variables

as part of the included filename, the approach uses a safe approximation that includes any file available to the application that matches the include expression.

3.2.2 Fitness Function

The fitness function used by the approach is similar to that used by Tracy et al. [TCMM98]. That is, for a predicate $a\ op\ b$ where op is a relational operator, fitness is zero when the condition is true and $|a - b|$ when the condition is false. A fitness of zero denotes the situation where the test vector assessed by the fitness function covers the desired branch. That is, the algorithm seeks to minimize fitness values throughout the search process.

Like Tracey et al., a value k (in this case 1) is added to penalize incorrect test data but the fitness function proposed here adds that value only in case of $<$, $>$ and $\neq$. The value 1 is added in the case of $<$, $>$ and $\neq$ to avoid an incorrect fitness of 0 to be assigned when the two operands are equal. For example, in a condition $if\ (a \geq b)$, where both a and b are equal to 5, the value of $|a-b|$ is 0 which is the correct fitness for this branch. However, in the case of a condition $if\ (a > b)$ where a and b are also equal to 5, the value of $|a - b|$ is 0 which incorrectly indicates that the branch is covered. By adding 1, the fitness reflects the fact that the branch was not covered but is very close to being covered.

For strings, the fitness function uses Levenshtein distance [Nav01], following Alshraideh and Bottaci [AB06]. The Levenshtein distance is the minimum number of insert, delete and substitute operations needed to convert one string to another string. The Levenshtein distance is suitable for $=$ and $\neq$ operators. For other operators, the fitness function converts the ASCII code of a string to a decimal representation and uses the same fitness used for normal numeric types following Zhao et al. [ZLM10].

Table 3.1 summarizes the fitness function calculations used in the algorithm proposed in this chapter. As a pre-processing step, compound predicates involving logical operators are expanded, using a pre-transformation, to simple relational predicates.

3.2.3 Test Data Generation Algorithms

The proposed algorithms for test data generation are all based on Hill Climbing using Korel's AVM [Kor90]. When a target branch is selected, AVM is used to mutate each input in turn while all other inputs remain fixed. When the selected mutation is

Expression	Numeric	String
$a = b$	$\|a - b\|$	*levenshtein(a,b)*
$a \neq b$	if $\|a - b\| = 0$ then 1	if *levenshtein(a,b)* $= 0$ *then* 1
	else 0	else 0
$a > b$	if $a > b$ then 0	if *Numeric(a)* $>$ *Numeric(b) then* 0
	else $\|a - b\| + 1$	else $\|$*Numeric(a)*$-$*Numeric(b)*$\|+1$
$a \geq b$	if $a \geq b$ then 0	if *Numeric(a)* $>$ *Numeric(b) then* 0
	else $\|a - b\|$	else $\|$*Numeric(a)*$-$*Numeric(b)*$\|$
$a < b$	if $a < b$ then 0	if *Numeric(a)* $<$ *Numeric(b) then* 0
	else $\|a - b\| + 1$	else $\|$*Numeric(a)*$-$*Numeric(b)*$\|+1$
$a \leq b$	if $a \leq b$ then 0	if *Numeric(a)* $<$ *Numeric(b) then* 0
	else $\|a - b\|$	else $\|$*Numeric(a)*$-$*Numeric(b)*$\|$

Table 3.1: Fitness calculations for Numeric and String variable types based on Tracy et al. [TCMM98] fitness function discussed in the literature review (Section 2.4.1). Levenshtein distance is the minimum number of insert, delete and substitute operations needed to convert one string to another string. Strings are converted to the numeric representation of their ASCII Code for relational operators. A constant 1 is added to the fitness for operators $>$ and $<$ to avoid assigning 0 fitness when the compared operands are equal.

Algorithm 1 NMS: Overall Test Data Generation Algorithm: Top level units are extracted from the File Tree Analyser results. Each unit is called with no parameters to get an initial 'work list' of reached branches. For each work list branch, the input vector is mutated iteratively until the branch is covered or the stopping criterion is satisfied. Near misses and collateral coverage are recorded for later use.

Require: Application Name *AppName*
Require: Static Analysis results *AnalysisDB*
$\mathcal{U}$: queue of top level file units to be processed. Retrieved from the File Tree Analyser results.
$\mathcal{B}$: queue of branches reached but not covered.
$\mathcal{C}$: Coverage table of all branches with the best achieved distance.
$\mathcal{T}$: Test cases that achieved best distance for each reached or covered branch.
$\mathcal{F}$: set of branch and fitness values achieved for the executed test case.
Input : setOf(inputname, value)
IV: Input Vector consisting of setOf(*Input*)
Distance: holds fitness value for a certain branch.

1: $\mathcal{U}$:= getTestUnits(*AppName*, *AnalysisDB*)
2: $\mathcal{T} := \emptyset$
3: **for all** U in $\mathcal{U}$ **do**
4: *IV*:= $\emptyset$
5: $\mathcal{F}$:=executeTestcase(U, *IV*)
6: $\mathcal{T}$:=updateTestdata($\mathcal{T}$,U,*IV*,$\mathcal{F}$)
7: $\mathcal{C}$:=updateCoveragedata($\mathcal{C}$, $\mathcal{F}$)
8: **while** first run or coverage improved **do**
9: $\mathcal{B}$:= getReachedBranches($\mathcal{C}$)
10: **for all** B in $\mathcal{B}$ **do**
11: initState()
12: *IV* := setInputVector(B, *AnalysisDB*,$\mathcal{T}$)
13: *Input* := NULL
14: *CurrentDistance* := getBranchDist(B, $\mathcal{C}$)
15: **while** *CurrentDistance* > 0 and not no improvements for 200 tries **do**
16: initilaizeDB()
17: *Input*:= mutateInputs(*IV*,*Input*,*CurrentDistance*, *PreviousDistance*)
18: *IV*:= replaceInputValue(*IV*,*Input*)
19: $\mathcal{F}$:=executeTestcase(U, *IV*)
20: $\mathcal{T}$:=updateTestdata($\mathcal{T}$,*IV*,$\mathcal{F}$)
21: $\mathcal{C}$:=updateCoveragedata($\mathcal{C}$, $\mathcal{F}$)
22: *PreviousDistance* = *CurrentDistance*
23: *CurrentDistance* = getBranchDist(B, $\mathcal{C}$)
24: **end while**
25: **end for**
26: **end while**
27: **end for**
28: **return** $\mathcal{T}$

found to improve fitness, the change in the same direction is accelerated. To avoid 'over shoot', when fitness is close to zero the approach decelerates.

The approach notes branches that it reached but failed to cover and targets them on subsequent iterations. That is, a branch is reached if its immediately controlling predicate is executed, while a branch is covered if the branch itself is traversed. This 'exploration' approach eliminates the need for calculating the so-called approach level [McM04]. This is because the approach attempts to cover a branch only when it is reached, i.e., all transitively controlling predicates on some path have been satisfied. The approach is called an 'exploration' approach because the technique maintains a 'current frontier' of reached but as yet uncovered branches, seeking to push back this 'frontier' at each top level iteration. A similar approach was used by Michael et al. [MMS01] for evolutionary testing.

At each iteration, the approach also keeps track of input values that cause any 'near misses'. A near miss is an input vector that causes fitness improvement for a branch other than the targeted branch. Near misses are used in place of random values when initializing a search to cover that branch. This approach will be referred to as 'Near Miss Seeding' (**NMS**).

More formally, Algorithm 1 describes the top level algorithm, which starts by calling the application with empty inputs for every top level file (Line 5). Every execution of a test case returns a list ($\mathcal{F}$) of all branches in that execution together with the distance achieved for them. This list is used to update a coverage table ($\mathcal{C}$) and the test suite ($\mathcal{T}$) for every branch that recorded an improvement in distance. A 'work list' of reached branches is extracted from the coverage table.

Every branch in the work list is then processed in an attempt to cover it. First, the state and database are initialized and the user (in this case the test tool SWAT) is logged into the application (Line 11). The input vector is then constructed using the analysis data. Values are initialized using the input values that caused the branch to be reached and random values for any additional inputs (Line 12). One input is mutated at a time and the new test case executed until the branch is covered or no improvements are possible.

Algorithm 2 describes the mutation process. If no input was selected for mutation or the last mutation did not affect distance, a new input is selected. If the last mutation

caused distance to increase, a new operator is selected. If the last mutation caused distance to decrease, the operation is accelerated. Finally, the selected input is mutated (Line 12). Algorithms 1 and 2 describe the unaugmented search based approach.

A few modifications are made to use constants collected from the source code of the application in the search process. Constants are used to initialize inputs in Line 12 of Algorithm 1 instead of random values and assigned to the input in Line 12 of Algorithm 2 when the input type is string. This approach was first proposed by Alshraideh and Bottaci [AB06] in the context of testing traditional applications. This process will be referred to as 'Static Constant Seeding' (**SCS**) thereafter.

A further modification of the algorithm is made to seed values dynamically mined during execution into the search space in a similar way to static constants. This process can be called 'Dynamically Mined Value' (**DMV**) seeding. More details about **DMV** are given in the next section.

3.2.4 Dynamically Mined Value Seeding

Constants collected statically are specific to the application and therefore can aid in covering branches that depend on these constants. In a similar way, collecting values dynamically from predicates can also prove beneficial. These collected values are not only specific to the application but also specific to the predicates from which they

Algorithm 2 Mutation Algorithm (mutateInputs): The algorithm determines the next input to which the search moves, based on distance achieved by the last mutation. It also decides when to change the mutation operator and when to accelerate the selected operation

Require: *IV*, *Input*, *CurrentDistance*, *PreviousDistance*, *AnalysisDB*
1: **if** *Input* is NULL or *CurrentDistance* = *PreviousDistance* **then**
2: *Input* := selectNewInput(*Input*,*IV*)
3: **else**
4: **if** *CurrentDistance* > *PreviousDistance* **then**
5: changeMutationOperator()
6: **else**
7: **if** *CurrentDistance* < *PreviousDistance* **then**
8: accelerateOperation()
9: **end if**
10: **end if**
11: **end if**
12: *Input*:= mutate(*Input*,*AnalysisDB*)
13: **return** *Input*

were collected. The proposed algorithm seeds these values into the search space when targeting their associated branches.

This approach can help the algorithm in covering branches faster especially when the branch predicate is directly dependent on an input. For example, the following branch is taken from an application used in the study (Schoolmate):

```
if ($_POST['page2']==1337) {
..}
```

When a standard search algorithm attempts to cover this branch, a random value for `page2` is chosen. The algorithm would then increase or decrease the random value to arrive to a value of `page2` that would cause the branch to be covered (in this case 1337). Using the value collected from this predicate, the algorithm would directly use the value 1337 thereby covering the branch faster. The value in `$_POST['page2']` would not be used because the approach keeps track of previously used values for each branch to avoid reusing them. In this case, the value was a constant and can be collected using static analysis. However, in other cases the two operands can be both variables making it useful to collect values at run-time.

Web applications offer a wealth of valid input values in their dynamically generated HTML. The output HTML is returned to the user (in this case the tool SWAT) in a structured form that makes it possible to extract these values and to associate them with their respective input fields. The source of these values is `form` data and embedded URLs. `Form` definitions can contain valid values in drop-down menus, check boxes, radio buttons and hidden values. Embedded URLs can also contain valid values in their query strings. These fields are populated from different sources that can include databases, configuration files and/or external data sources. The input fields associated with these values can affect coverage indirectly or through hard to cover branches. An example that illustrates the potential of **DMV** is the following predicate taken from one of the applications (PHPSysInfo) used in the empirical study:

```
if (file_exists($lng.'.php')) {
..}
```

Generating a value for input `$lng` that would cover the `true` branch of the control statement might be hard since the condition is a flag condition that would not provide

much guidance to the search process. However, a `form` in the dynamically generated HTML (Figure 3.1) has a drop-down menu that contains a list of language options available for the application (i.e. the language file exists in the application's file system). Using one of the values in the drop down menu for this input field would cover the branch.

The approach mines the HTML returned when executing test cases (Line 19 of Algorithm 1) to collect such values and subsequently seed them into the search when mutating the inputs associated with them.

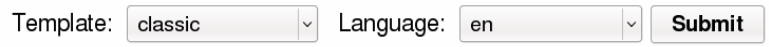


Figure 3.1: Form taken from PHPSysInfo

3.3 The SWAT Tool

A tool called the 'Search based Web Application Tester' (SWAT) was developed to implement the approach and embed it within an end-to-end testing infrastructure. SWAT's architecture is illustrated in Figure 3.2. The tool is composed of a pre-processing component, the Search Based Tester and the Test Harness.

The original source code is passed through the Predicate Expander and Instrumenter. This produces a transformed version of the code where predicates with logical operators are expanded and control statements are instrumented to calculate fitness in addition to the predicates' original behaviour. The code is also instrumented to collect run-time values to be used in subsequent Dynamically Mined Value seeding.

The Static Analyser performs the analysis needed to resolve the issues mentioned in Section 3.2.1. The results are stored in the Analysis Data repository and used later by the Search Based Tester (SBT). The Constant Extractor mines the code for constants to be used in subsequent Static Constant Seeding. The Input Format Extractor analyses the code to extract the input vector. The File Tree Analyser generates a tree in which nodes denote files and edges denote include relationships. This information is used to determine the top level test units to be processed.

The Input Type and Login Discoverer component performs a simple combination of static and dynamic analysis to infer input types and to identify the login process.

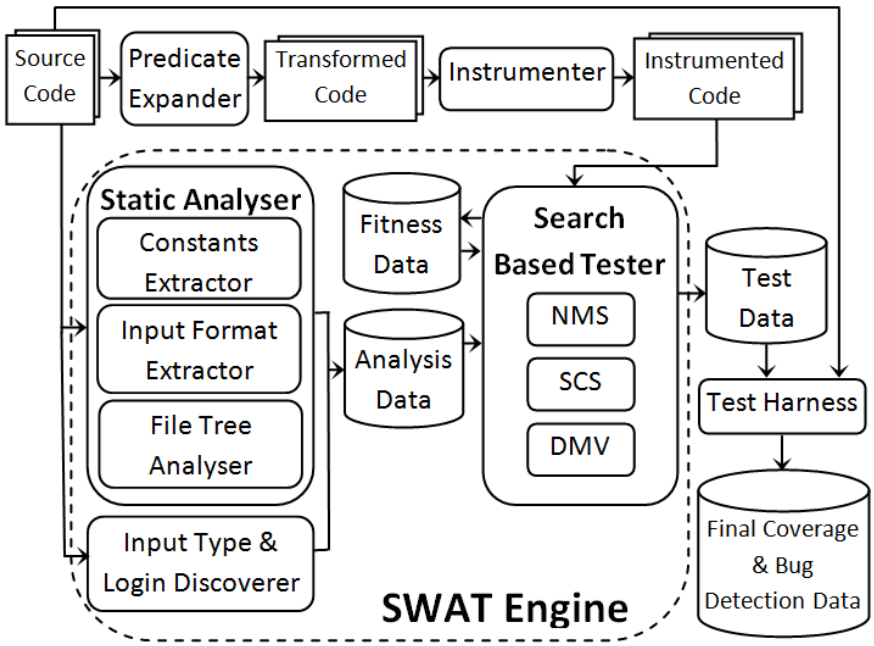


Figure 3.2: SWAT tool architecture

This is the only component for which results need to be augmented manually; this is because the technique for type inference is unable to infer types for all inputs. The Login Discoverer is used to dynamically extract the variables used to store the username, password, login URL and any other inputs that need to be set for login. This is achieved by performing a quick crawl of the application to locate the login `form`. A login `form` is identified by the presence of one field of type 'password'. A login `form` is expected to have one password field while a `form` to create an account is expected to have two password fields to reconfirm the selected password. The concrete values for username and password are provided to the tool.

Stratego/xt [BKVV08] and PHP-Front [BB10] were used to develop the Predicate Expander, the Instrumenter, the Static Analyser and the static analysis part of the Input Type and Login Discoverer. Stratego/xt is a program transformation language and PHP-Front provides libraries for Stratego/xt supporting PHP. The Input Format Extractor was taken from the PHP-Front project with minor alterations. All other static analysis and transformation tools have been developed from scratch. The dynamic part of the Input Type and Login Discoverer was developed using Perl and Java. The data produced from the Static Analyser are uploaded to a MySQL database using a tool developed in Java.

The Search Based Tester (SBT) uses the transformed source code and the analysis data and implements the input generation technique described by Algorithms 1 and 2 and the augmentations needed for **SCS** and **DMV**. When a test case is executed, the instrumented application writes fitness information and values collected dynamically for seeding to a file that is later accessed by the SBT. An earlier version of the tool used a database table to communicate these values to the SBT. However, the overhead of database transactions caused significant degradation in the tool's performance. The current version writes information to a file while maintaining a list of best recorded fitness for each branch on that specific execution. If a new fitness does not improve on the best recorded fitness so far, the file writing operation can be avoided.

The Test Harness uses the generated test data to run the tests on the original source code and to produce coverage and bug data. When a test case is executed, the generated HTML together with the web Server's error logs are parsed for PHP execution errors. The Search Based Tester and Test Harness are implemented in Perl and use the HTTP, HTML and LWP (Library for WWW in Perl) libraries.

3.4 Evaluation

For the evaluation, three versions of the tool SWAT were implemented. Each version adds one of the enhancements described in Section 3.2.3 in the following way:

- **NMS** implements the Near Miss Seeding unaugmented approach described in Algorithm 1 in Section 3.2.3.
- **SCS** is **NMS** with Static Constant Seeding.
- **DMV** is **SCS** with Dynamically Mined Value seeding.

Each branch was allocated the same budget of fitness evaluations for each version of the tool. In this way the effects of each of the proposed enhancements on the unaugmented traditional search based approach can be evaluated.

The experiment was designed to answer the following research questions:

RQ1: How does each of the proposed enhancements affect branch coverage?

To answer this question, the evaluation compares branch coverage for each of the algorithms. Coverage was measured on the original untransformed application. The

original application was instrumented to record coverage without the transformations to expand predicates and calculate fitness.

RQ2: How does each of the proposed enhancements affect efficiency of the approach?

To answer this question, the number of fitness evaluations needed per application and per branch were calculated. The evaluation also reports the elapsed time and CPU time used in testing per application and per branch.

RQ3: How does each of the proposed enhancements affect fault finding ability?

To answer this question, the automated oracle described in Section 1.5.2 was used to compare the fault finding ability of the test suites produced by each enhancement. Wilcoxon unpaired one-sided signed rank test at the 95% confidence level was performed to determine the statistical significance of the observed results.

3.4.1 Experimental Set-up

The evaluation ran each of the three versions of the tool 30 times on each of the six PHP applications and collected coverage data. Data for repeated runs is provided in order to cater for the stochastic nature of the search based optimization that lies at the heart of the presented approach. Multiple runs are samples from the space of all possible runs. With 30 runs of each algorithm, a sufficient sample size for statistical significance testing is provided. The tool was executed on an Intel Core 2 Duo CPU, running at 2 GHz with 2 GB RAM.

The applications studied (described in Section 1.5.1) were installed and set-up locally on the same machine used to generate inputs. The database for each of these applications was set up following an identical systematic strategy as follows: All tables are populated in a minimal manner, but such that each table contains at least one record and a record is created for each possible value of a column of an enumerated type. All applications except one (PHPSysInfo) use a database. For configurable applications, all features were enabled where the application permitted it. Where login is required to use the web application, a valid username and password pair was supplied to the tool.

3.4.2 Branch Coverage

Table 3.2 summarizes the results obtained by the experiment. Coverage results are reported together with the number of test cases generated to achieve this coverage.

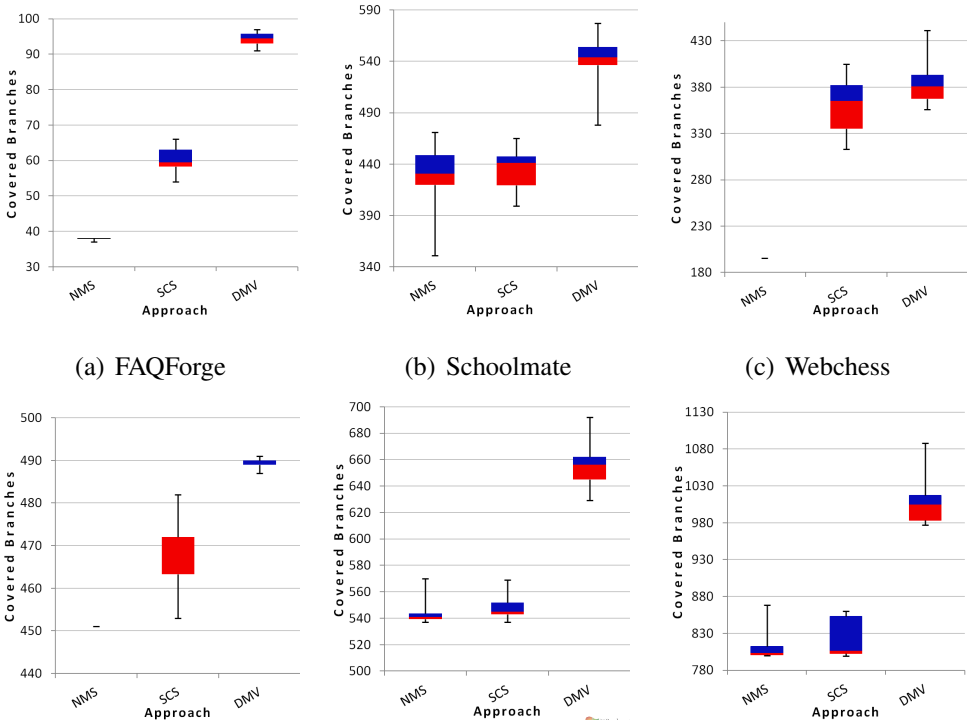
Table 3.2: Average coverage and execution time results obtained by running each algorithm 30 times for each application with the same budget of evaluations per branch for each version. Effort is the number of evaluations per branch covered. Results in bold are statistically significantly better than the results above them using the Wilcoxon's test (95% confidence level).

App Name	Alg	Total #branches	Test cases	Fitness Evals	Covered Branches Num	Covered Branches %	Covered Branches Effort	Elapsed Time Time	Elapsed Time per Br	CPU Time Time	CPU Time per Br	Average Faults Found Crash	Average Faults Found Error	Average Faults Found Warning
FAQForge	NMS	142	25	4809	38.0	26.7	126.7	104	2.75	28	0.734	0	0	20.0
	SCS		22	1453	**60.2**	**42.4**	**24.1**	40	**0.66**	16	**0.266**	0	**1.0**	**25.0**
	DMV		34	2177	**94.4**	**66.5**	**23.1**	64	0.69	23	0.249	0	**5.9**	**46.1**
Schoolmate	NMS	828	164	21840	428.9	51.8	50.9	949	2.21	228	0.531	2.1	13.5	75.5
	SCS		167	19037	435.5	52.6	**43.7**	712	**1.63**	229	0.525	**2.6**	**15.7**	75.5
	DMV		172	12641	**542.3**	**65.5**	**23.3**	549	**1.01**	211	**0.388**	**3.6**	**21.5**	**87.4**
Webchess	NMS	1051	21	9542	195.0	18.6	48.9	705	3.62	41	0.209	0	6.1	9.0
	SCS		43	10650	**360.9**	**34.3**	**29.5**	922	**2.55**	83	0.230	0	**16.9**	**41.7**
	DMV		45	8953	**382.6**	**36.4**	**23.4**	879	**2.30**	82	0.215	0	**20.0**	**55.3**
PHPSysInfo	NMS	1451	8	1529	300.0	20.7	5.1	5891	19.64	13	0.043	0	0	3.0
	SCS		11	1398	**315.4**	**21.7**	**4.4**	5302	**16.81**	12	0.039	0	**2.9**	**3.9**
	DMV		20	1337	**333.4**	**23.0**	**4.1**	4459	**16.37**	53	0.160	0	3.2	**4.0**
Timeclock	NMS	3567	116	7212	543.6	15.2	13.3	1083	1.99	56	0.103	0	0	155.0
	SCS		248	8445	**548.5**	**15.4**	15.4	1135	2.07	58	0.106	0	0	**155.9**
	DMV		244	12239	**655.3**	**18.4**	19.4	754	**1.15**	77	0.117	0	**1.6**	**173.2**
PHPBB2	NMS	5680	116	24690	816.6	14.4	30.2	5956	7.29	259	0.317	0	3.0	41.1
	SCS		248	22981	821.6	14.5	**28.0**	5533	**6.73**	251	0.306	0	3.0	**41.8**
	DMV		244	24080	**1007.3**	**17.7**	**23.9**	5821	**5.78**	252	**0.248**	0	**4.6**	**58.4**

The number of covered branches increases with **SCS** for all applications. Webchess displays the highest improvement with an 85% increase in coverage. Overall, **SCS** recorded an average increase in coverage of 25.1% compared to **NMS**.

Covered branches also increase with **DMV** for all applications studied. An average improvement of 22.3% was observed in branch coverage over all applications with FaqForge showing the highest improvement.

Figure 3.3 shows the variation in branch coverage achieved over 30 runs of each approach. We notice that **DMV**'s lowest coverage is higher than the highest coverage of other approaches for five of the six applications. For three of the applications **NMS** shows little or no variation over 30 runs. For all applications studied a statistically significant increase in branch coverage at the 95% confidence level for **DMV** was observed compared to **SCS** and **NMS**.

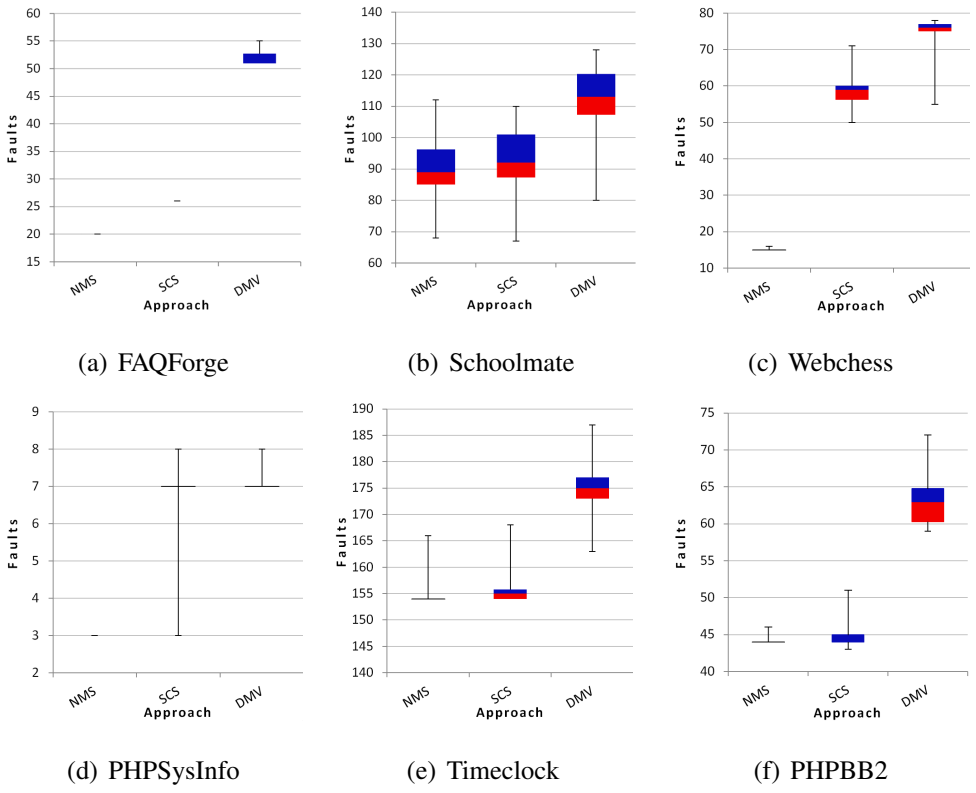


(a) FAQForge (b) Schoolmate (c) Webchess

(d) PHPSysInfo (e) Timeclock (f) PHPBB2

Figure 3.3: Coverage results over 30 runs of each of the three algorithms on each of the six web applications.

In three applications, running the same test suite twice can produce different branch coverage levels on each occasion. Eliminating this non-determinism is not



(a) FAQForge (b) Schoolmate (c) Webchess

(d) PHPSysInfo (e) Timeclock (f) PHPBB2

Figure 3.4: Fault results over 30 runs of each of the three algorithms on each of the six web applications.

desirable because it manifests important aspects of the application. Investigating the branches covered revealed the following observations: When a new game is initiated in Webchess, the user can either choose his/her preferred colour or choose random. A similar issue is found in PHPSysInfo where the user can select the template of the output or select random. Timeclock has a weather display feature that, when enabled, fetches the current weather information from the internet and displays it. The current weather information affects coverage of the branches that control the format in which this weather information is displayed.

3.4.3 Efficiency

To measure efficiency, average execution times were recorded and effort measured for each approach over 30 runs. The reported CPU time does not include time spent calling and processing the PHP Application (which the reported elapsed time does). Effort is the ratio between total number of fitness evaluations and branches covered. Effort can be considered to be the more reliable empirical assessment of the search algorithm

time complexity since it is unconfounded with difficulties of measuring time in a multi process environment. However, figures for elapsed time are more useful as a rough guide to likely overall test data generation time performance for each application, which is affected by the system under test as well as the performance of the algorithms.

Effort and time results are reported in Table 3.2. Effort decreases with each of the two algorithmic enhancements for all applications except Timeclock, for which effort increases. Overall six applications studied, **SCS** decreased effort per covered branch by an average of 11.2% while **DMV** decreased effort by a further 30.17%.

FAQForge is processed the fastest (1 min) while PHPBB2 is the slowest with an average of approximately 97 minutes per run for **DMV**. Of course, SWAT is only a research prototype. However, even this worst case execution time can be accommodated within a daily build cycle, with overnight automated test data generation.

3.4.4 Fault Finding Ability

Table 3.2 reports the average numbers of faults found for each algorithm and each application over 30 runs. Overall, each enhancement increases the number of faults found for each of the fault categories. **DMV** performs best in fault finding for all six applications. In almost all cases, with the exception of faults of type error for PHPSysInfo, the improvement is statistically significant. Schoolmate is the only application where crashes were found.

Figure 3.4 shows the variation in total faults found for each application over 30 runs. Some of the faults found indicate a lack of adequate validation of inputs before use in critical operations, such as database queries. Inputs are concatenated directly to SQL statements which could cause a security threat. In some cases, where faults were parsed from the generated HTML, the SQL statement that caused the error was displayed to the user, giving hackers the opportunity to analyse the statements and form an SQL injection attack. PHPBB2 uses an input field to redirect requests to other parts of the site (e.g., after login). This input field is displayed to the user (in the query string) and can be modified to potentially gain access to server files. Other faults found include inclusion of non-existent files and wrong use of functions.

3.4.5 Answers to Research Questions

This section answers the research questions posed at the start of this section, based on the empirical evidence from the experiments on the six web applications.

RQ1: How does each of the proposed enhancements affect branch coverage?

The results of the evaluation provide evidence to support the claim that each enhancement improved branch coverage for all of the six applications under test. In particular, it can be noted that **DMV** statistically significantly outperforms **NMS** and **SCS** for all six applications studied.

SCS also, in turn, achieved higher coverage than **NMS** for all six applications. Wilcoxon's test indicated these results to be statistically significant for four of the six applications. A closer look at the type of branches that were additionally covered by **SCS** reveals that they are predominantly string predicates that involve constants. In Schoolmate and PHPBB2, the improvement in branch coverage was not statistically significant. This can be attributed to the fact that those two applications have relatively fewer constant-using string predicates. The average percentage of predicates that involve a string constant overall applications is 27% while for Schoolmate, for example, the average is only 15%.

Branch coverage results for **DMV** compared to **SCS** statistically significantly increase for all six applications. By analysing additionally covered branches, it is observed that **DMV** performs better than **SCS** at covering constant-using string predicates. It also covers string predicates that are variable-using rather than constant-using.

As might be expected, both algorithms appear to achieve higher coverage when the application contains key string predicates that uncover unexplored parts of the application under test. The empirical results suggest that this situation may be sufficiently common for seeding to be very effective.

RQ2: How does each of the proposed enhancements affect efficiency of the approach?

Effort decreases statistically significantly by using **SCS** rather than **NMS** for all applications except one. Timeclock is the only application were effort increased instead

of decreasing. This may be caused by the nature of constants mined from Timeclock: Float constants mined from Timeclock had a precision as high as 16 decimal places while the highest for all other applications was three decimal places. Assigning these constants to input variables when initializing the input vector may have, in some cases, not assisted and possibly even impeded the search process for Timeclock.

Using **DMV** caused effort to decrease for all applications except one. Wilcoxon's test indicated that this reduction was statistically significant in the five cases where the reduction was observed. Like **SCS**, effort for Timeclock increased with **DMV**. This could be caused by the fact that the applications were transformed before running the tool to decompose 'And' and 'Or' statements. However, coverage is measured on the untransformed version. The transformation is merely an enabling testability transformation [HHH+04] and it would be unreasonable to attach any importance to coverage of an internal representation. However, this does account for the increase in effort for Timeclock: While for the untransformed version of Timeclock the effort for **NMS** is 13.3 increasing to 19.4 for **DMV**, for the transformed version of Timeclock this effort reduces from 13.0 for **NMS** to 12.4 for **DMV**.

RQ3: How does each of the proposed enhancements affect fault finding ability?

The number of errors and crashes statistically significantly increased by using **SCS** for all applications where errors and crashes were found. The same is observed for warnings for all applications except Schoolmate. This may be tied to the fact that coverage for Schoolmate using **SCS** does not increase statistically significantly.

All error types record an increase in numbers when using **DMV** for all applications where faults are found. Wilcoxon's test indicates that this increase is statistically significant for five out of six applications.

3.4.6 Threats to Validity and Limitations

Internal threats: The internal threats that could affect the validity of results depend on the set-up of the applications. Results could be affected by database state and the application's configurations. To minimize bias, a systematic procedure was defined for populating the database and configuring the application. This procedure ensures that

no prior knowledge about the applications under test can be exploited and is performed in the same manner for all applications.

External threats: External threats are related to the choice of applications and the degree to which one can generalize from the results obtained from those chosen for the study. The applications were selected to provide compatibility with previous research on testing web applications. However, they are real applications used by real users as the high number of downloads from SourceForge indicates.

Steps were taken to ensure that reported results would be reproducible. The state of the application was initialized before each test case is called. The applications used are open-source and thus publicly available. Bug reports are available online.[2]

Limitations: The overall aim is to produce a fully automated testing approach that generates tests, runs them and reports faults found entirely automatically. However, there are some aspects of the overall approach that are not, as yet, fully automated. Deciding input types is partially manual. Username and password information also needs to be provided by the user. Some data types used in predicates are not yet supported by the Instrumenter. Enhancing the tool to handle all data types and defining better fitness functions for arrays and objects may further improve coverage.

3.5 Related Work

This chapter applied search based testing to web applications and introduced using Dynamically Mined Value to the search process. The work presented in this chapter was also imbued with ideas collected and adopted from several previous approaches in the SBST literature that were previously discussed in Section 2.4 of the literature review. SWAT retains the Alternating Variable Method (AVM) introduced by Korel [Kor90] in adopted form. SWAT's approach to search exploration adopts a similar systematic technique for branch order to that used by Michael et al. [MMS01] for C programs using evolutionary algorithms. Michael et al. also keep track of inputs that caused the branch to be reached to use as seeds. However, the overall algorithm and application domain, being stand-alone C applications, was very different to the one presented here. Seeding constants gathered from the source code to the search space and using the Levenshtein distance to measure fitness for strings was first proposed by

[2] `http://www.cs.ucl.ac.uk/staff/nalshahw/swat`

Alshraideh and Bottaci [AB06]. SWAT used the same idea but applied it to larger scale web applications implemented in PHP.

Halfond et al. [HAO09, HO08] introduced an algorithm that uses symbolic execution of the source code to group inputs into interfaces. The approach was applied to Java applications, while the approach presented in this chapter is applied to PHP applications. Wassermann et al. [WS07, WYC+08] also used symbolic execution to generate test data for web applications. However, their work focused on SQL injection attacks and examined only functions that call database queries. More details about these two approaches were provided in the literature review in Chapter 2 on pages 43 and 47.

Artzi et al. [AKD+10] automatically generated test cases for dynamic web applications using Dynamic Symbolic Execution by developing the tool Apollo. The details of the approach were discussed on page 48 of the literature review. Their approach also targeted PHP applications. However, the two approaches differ in the test adequacy criteria (statement coverage vs. branch coverage). Their approach also produces a different number of test cases since all test cases generated during the run of the tool are collected. Their algorithm minimizes the test suite in regards of faults found using an automated oracle (similar to the one used in the evaluation and described in Section 1.5.2) for fault localization.

To perform a fair comparison of SWAT and Apollo, both tools have to be executed in the same environment and with the same initial state. More importantly, both tools need to have the same goal, for example maximizing branch coverage. A comparison of the current implementation of the tools would always result in test suites for Apollo that are several orders of magnitude larger than those obtained for SWAT. That is, Apollo is designed to capture all test cases that reveal an automated fault while SWAT only retains test cases that contribute to branch coverage. The automated oracle is used in SWAT to evaluate and compare algorithms however the aim is to produce a test suite that maximizes branch coverage.

Section 2.3.2 reviewed bypass testing of web applications. In the work presented in this chapter, test cases generated by SWAT also ‘bypass’ the interface to submit data to the server-side code directly. The test data, although not specifically generated for bypass testing, could be used for that purpose.

3.6 Conclusion

This chapter introduced a set of related search based testing algorithms, adapted for web application testing and augmented with static and dynamic seeding. It also introduced a tool, SWAT, that implements the proposed automated test data generation approach for PHP web applications. SWAT draws on more than ten years of results reported for search based testing, as applied to conventional stand–alone applications, seeking to exploit and build upon best practice and proven results where possible. However, as the chapter shows, there are many issues raised by web application testing, such as dynamic type binding and user interface inference that create novel challenges for search based testing that have not previously been addressed. The chapter also shows how a novel Dynamically Mined Value seeding approach can significantly reduce effort and increase effectiveness for web application testing.

The chapter reports on an empirical study that evaluates the approach on six PHP web applications ranging in size up to 20k LoC, presenting results concerning coverage, various measures of test effort and also an analysis of fault detection ability. SWAT detected an average of 60 faults and 424 warnings over all six applications studied.

Chapter 4

State Aware Test Case Regeneration

4.1 Introduction

The previous chapter used the client-side output to enhance the generation of test data for structural coverage of the server-side web application code. This chapter uses the server-side output to enhance the generation of test sequences to test a web application on the navigational level. Specifically, the Def-Use pairs of session variables and database tables and values maintained in them are used to generate test sequences that are more likely to exercise new parts of the application. The test suites generated by SWAT in Chapter 3 are used as an input to the approach proposed in this chapter. The work in this chapter is concerned with generating effective sequences of requests. For sequences to be effective, the input values used in each individual request need to exercise different application behaviour. Therefore, SWAT test cases, which are generated to maximize branch coverage, could be a potentially good candidate for this purpose.

Web application developers have little control over how their applications are used due to browser functions, such as `Back` and `Refresh`. This can lead to unexpected execution paths that may cause unanticipated behaviour. An analysis by Ricca and Tonella [TR04b] found that more than 40% of user sessions contained paths that are considered 'infeasible' by the application model, but which are, nevertheless, achievable in practice, using browser functions.

The order in which requests are supplied to the server affects the behaviour of the application. This order can also have a crucial influence on the degree of coverage and faults detected during testing. When Sprenkle et al. [SGSP05b] replayed test

requests multiple times in random orders without resetting the application state between requests, they observed increased server-side coverage and elevated fault detection.

These findings suggest that additional testing value can be added to a test suite, by execution of multiple reordered request sequences. However, exhaustive exploration of all possible such reorderings will be prohibitively expensive, even for small test suites. Therefore, to get additional value from request orderings an intelligent algorithm needs to be developed for recombination of requests. The sequences generated must be likely to achieve increased coverage and fault detection without encountering an exponential explosion in the number of sequences to be executed.

To achieve this, this chapter seeks to use server-side dataflow analysis of the server state to guide the choice of request orderings that increase effectiveness. Of course, the Hypertext Transfer Protocol (HTTP) itself is stateless; the server processes each request independently without requiring any knowledge of previous requests from the user. The overall state of the web application is thus maintained between multiple HTTP requests using other techniques, such as session variables, cookies, hidden variables and the database.

Cookies and hidden variables are stored on the client-side, making it possible for the tester to force either of these two state carriers to contain arbitrary values of choice. Faults exposed by setting either a cookie or a hidden variable to some chosen value are, by definition 'real' faults, because there is nothing to prevent the user of the application from setting either in just this manner. However, on the server side, things are different: The only way that the user can set a session variable or a database table to a particular value, is through the execution of client-side requests that cause the server to execute some part of its code that affects these two forms of state carrier.

The tester could artificially insert values into database tables or into session variables, but there would then be no guarantee that any faults detected by such artificial value insertion would be true positives; perhaps there is no client-side request sequence that can create such values. Therefore, the approach proposed in this chapter seeks request sequences that cause the server to exercise the server-side web application state.

A new technique is introduced for generating new sequences of HTTP requests from an existing test suite of requests. The technique is inspired by dataflow testing, specifically seeking to execute a definition of a state variable (a session variable or

database table) and to ensure that this value flows unchanged to a corresponding use. Though dataflow testing is well understood in conventional applications, there has been no previous work on state-based dataflow testing of web applications.

This chapter introduces a Def-Use (DU) approach that seeks to ensure that each definition flows to each use. It also introduces a novel form of dataflow testing, value-aware dataflow testing, which seeks to ensure that each possible different value obtained at a session variable flows into each use and that each different dynamically generated database invocation reaches a corresponding use. A tool, SART (State Aware Regeneration Tool), is introduced that implements this technique and is used to experiment with the approach.

This chapter also reports the results of an empirical study on four real world web applications for the proposed state-based value-aware dataflow technique and also, as a baseline comparator, for random recombination. All experiments start with high quality test suites drawn from those whose cases alone, when executed individually, already achieve good coverage and fault detection. High quality suites are chosen in order to test the ability of the recombination to add value; clearly if the starting suites were low-coverage request sets, then any recombination can be excepted to produce additional coverage and may thereby be sufficiently fortunate to reveal additional faults. However, the evaluation shows that the proposed approach can statistically significantly improve coverage and fault detection for relatively high quality test suites.

The primary contributions of this chapter are as follows:

1. The first application of server-side state-based dataflow testing techniques to web applications for test sequence regeneration.

2. The introduction of a novel value-aware DU approach that is sensitive to the specific state instance: session variable values and database SQL statements and a tool that implements it.

3. An empirical evaluation of the approach on four real-world web applications that confirms the effectiveness and efficiency of this approach. The evaluation shows that an average improvement of up to 25.31% can be obtained for branch coverage and 14.31% for fault detection.

The rest of this chapter is organized as follows: Section 4.2 provides a background concerning web application state. Section 4.3 introduces the proposed approach, whilst Section 4.4 introduces a tool called SART that implements the approach. Section 4.5 presents the evaluation together with a discussion of the results. Section 4.6 presents related work and Section 4.7 concludes.

4.2 Web Application State

In web applications, several techniques that propagate the state to subsequent requests are used to overcome the stateless nature of HTTP requests. These techniques include the use of session variables, client-side cookies and hidden `form` variables. In applications that use a database, the database state can also affect request behaviour.

This chapter focuses on PHP server-side code for concreteness, but the techniques introduced can be applied to other server-side languages.

4.2.1 Server Session Variables

In PHP, session variables are created by the server for a single user session and maintained until the session is terminated. Session variables are saved in a global array (`$_SESSION`) which can be accessed and modified by the server-side code:

```
$_SESSION['Var'] = value;
```

The index of the array is the variable name while the array element holds the value. The variables in the session hold their values for the duration of the session: when a variable is set, it can be accessed in any HTTP request that is submitted by the same user until the session is terminated and the session array is destroyed. A common use-case example is a session variable flag that determines the state 'logged in'. Such a flag is used to record whether a user has logged in to prevent execution of any requests prohibited to non-logged-in users.

In addition to the global array `$_SESSION`, built-in functions are used to modify and/or use the global session array as a whole. For example, the functions `start_session()` and `destroy_session()` create and destroy a session (and its associated session array) respectively. A complete list of session functions can be found on the PHP language website.[1]

[1] `http://php.net/manual/en/ref.session.php`

4.2.2 Database State

The database is an integral part of many web applications' operation. For example, online shopping applications use a database to keep track of their inventories, while social networks store user profiles and interactions.

A PHP program manipulates the database state through an API; a library is provided that includes functions to connect to the database and to execute SQL statements. For example, a simple SQL query to a MySQL database can be executed by calling:

```
$result = mysql_query("SELECT * FROM TABLE");
```

SQL statements are treated as strings in the native PHP code and can be constant or dynamically constructed (and therefore will have values that depend upon user inputs and conditional paths through the program). Many other web languages (and non-web specific languages) use the same approach.

The database state could affect the way a request is processed, resulting in different behaviour in response to the same request when executed with different database states. An example of this is a request to create a user account. The first time the request is executed, the application accepts the request and creates the account, but subsequent requests will, of course, be rejected.

4.3 Approach

For this chapter's purposes, a test suite will be considered to be a set of sequences of HTTP requests to the server-side issued at the client-side. The proposed approach seeks to take an existing test suite and to regenerate it. That is, to combine fragments of the request sequences to achieve improved coverage and fault detection. The starting test suite from which the approach regenerates can come from any existing approach [AH11, AKD+10, HO08, RT01a, WYC+08].

Ideally, regeneration should try every possible order and combination of all requests in the test suite. However, this is likely to be infeasible even with small test suites, because the number of possible sequences grows exponentially.

In order to focus on a manageable yet valuable subset, this chapter proposes an approach that generates test sequences by combining HTTP requests that define and

use the server-side state. The approach shares the same principles embodied in existing dataflow testing approaches [LK83, RW85]. However, the Def-Use approach proposed is defined at the page level, represented by an HTTP request, rather than the statement or block level. A Def-Use pair (DU pair) is a sequence where one test case (or request) defines the state and one uses the state. Statement locations are used to identify distinct definition and use points. Dataflow testing principles are employed and augmented to generate test sequences that are more likely to enhance the effectiveness of the original test suites in both coverage and fault finding.

The approach is to seek HTTP request sequences that cause server-side code to be executed to define the value of a session variable or database table. The approach then appends to this definition sequence an extra request that causes the server-side code to execute a corresponding use of the session variable or database table. This is a variation of standard Def-Use (DU) testing for web application server-side state. The main difference being the focus on session variables and database tables and the need to execute a sequence of requests to activate server-side definitions and uses.

A value-aware enhancement to the standard DU approach, that is more fine grained, is also introduced; it considers each different value defined and used not merely each Def-Use pair. In a conventional application this would be simply impractical because there would be too many different values to consider. However, as the empirical results (Section 4.5) shall show, for web applications this approach is not only feasible, but it produces a significant improvement in coverage and fault detection.

In the remainder of this section the proposed state-based DU approach and the value-aware enhancement is described in more detail.

4.3.1 State-based DU

This section describes the basic principles of the proposed State-based DU approach while the next section describes the modifications to this approach that are needed for the value-aware DU approach.

The approach proposed is based on a standard DU approach but adapted to web application state. For every distinct DU pair of session variables and database tables, a test sequence is constructed that covers each pair from the available HTTP requests in the test suite.

First, the approach needs to determine locations in the code at which session variables or database tables are defined or used. For session variables, a definition is an assignment and a use is any other reference to the variable, much as definitions and uses are constructed in non web applications. However, unlike non-web-based systems, a Def-Use pair can only be executed by issuing a sequence of requests from the client-side. Session functions are classified into functions that define the state and functions that use the state based on their descriptions. Because most session functions operate on the session as a whole and not on specific variables, when constructing sequences to cover DU pairs that are caused by session functions, the session array is considered to be the session variable.

For the database, `UPDATE`, `DELETE` and `INSERT` statements are considered to be definitions, while `SELECT` statements are considered to be uses.

The next step is to dynamically analyse the original test suite to match its test cases to the definition and use locations that were discovered by the static analysis. Each test case is executed on an instrumented version of the application to discover which definition or use locations it executes.

Algorithm 3 Test sequence generation approach for State-based DU. The state identifier (SI) refers to a session variable name or a table name. Test sequences are generated for every DU pair for each session variable and database table.

Require: Test Suite *TS*
Require: State identifiers *SI*
Require: *SI* definition locations *SIDL*
Require: *SI* use locations *SIUL*

1: $TS' = \emptyset$
2: **for all** *si* in *SI* **do**
3: **for all** *defloc* in *SIDL* **do**
4: *deftest* = getdeftestcase(*TS*,*si*,*defloc*)
5: **for all** *useloc* in *SIUL* **do**
6: *usetestid* = getusetestcase(*TS*,*si*,*useloc*)
7: **if** *usetestid* **then**
8: *newSeq* = (*deftest*,*usetestid*)
9: $TS' = TS' \cup newSeq$
10: **end if**
11: **end for**
12: **end for**
13: **end for**
14: **return** TS'

Algorithm 3 describes how test sequences are generated. A state identifier (SI) refers to a session variable name or database table name. The algorithm uses the analysis data to construct new test cases using the HTTP requests in the original test suite. For every state identifier, all definition locations are retrieved from a State Identifier Definition Locations (SIDL) set created during the static analysis phase. A test case that covers that definition location is then retrieved (Line 4). For all use locations of that same identifier, an HTTP request that executes that location is used, if one exists, to construct a new test sequence (Lines 5-9). The output of the algorithm is a set of new test sequences (TS′).

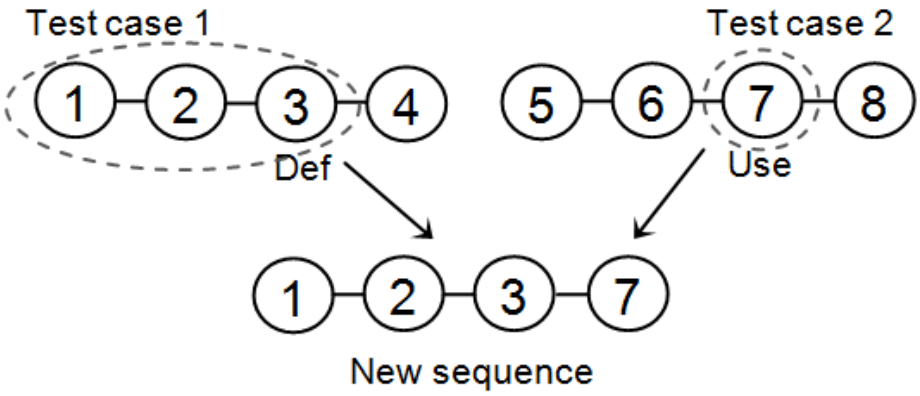


Figure 4.1: DU Sequence construction technique: The sequence that contains the definition HTTP request is truncated after the request and combined with the use HTTP request to form the new sequence.

When constructing a sequence, the algorithm first finds a test case that contains an HTTP request that defines the state identifier. When the required test case is found, the part of the test case leading to the definition HTTP request is extracted to form the first part of the new sequence. This maintains all requests that are required for the definition to be executed. The algorithm finds an HTTP request that uses the state identifier for every use location (if at least one exists; if none exists then there is no DU pair). New sequences are generated by applying the use HTTP request to the definition sequence. The use test cases are chosen so that the test case traverses a definition-clear path (if one exists). This is ensured by only selecting use requests that do not also execute a definition of the same session variable or database table. Figure 4.1 illustrates how sequences are generated.

Identifying Database Def-Use Pairs

A simple static analysis is used to identify session variables' definitions and uses. However, identifying database tables' definitions and uses might be more challenging. The approach needs to identify the locations of database calls within the code as well as the type of operation (`UPDATE`, `DELETE`, `INSERT` or `SELECT`) and the affected table. Calls to the function `mysql_query` in the program signify a database operation. However, to identify if the call is a definition or use and to identify the affected table, the SQL statements used at each call need to be collected and parsed to extract table names and operations.

SQL statements that manipulate the database are treated as regular strings in the PHP code. These strings can be constructed dynamically. For example, a code fragment that implements a database call from FAQForge, one of the subject applications used in the evaluation, is:

```
$q="UPDATE FaqPage SET page_num=$new_num WHERE ";
$q .="page_num=$page_num AND owner_id=$id";
$result=mysql_query($q,$dbLink);
```

In this example, the SQL statement `$q` is constructed by concatenating the constant strings and the user-provided inputs and dynamically computed variables. It is possible to approximate the SQL statement by using constant propagation and removing the dynamic parts of the statement. The approximated string might not be a valid SQL statement, but might contain sufficient information to identify the table name and SQL operation.

For example, for the code fragment above, the approximated string would be:

```
"UPDATE FaqPage SET page_num= WHERE
  page_num= AND owner_id="
```

From this string the approach can automatically extract the operation `UPDATE` and the affected table `FaqPage` and conclude that this is a definition of table `FaqPage`.

However, there are still cases where static analysis is unable to be accurate or complete. If the table name is constructed dynamically, for example, by reading a prefix from a database table, static analysis will not be able to identify the table's name. In other cases, centralized methods or functions are used to handle all database calls with

the query constructed and passed to the function at run-time. In such cases, the benefits of a Def-Use analysis might be very limited since the centralized functions would result in only a few calls to the function `mysql_query`. For example, if one function is defined to handle all update operations for all tables and another function is defined to handle delete operations and so on for all other SQL statement types, the number of Def-Use pairs for the application will be very small and affected tables cannot be identified statically. Due to these reasons, a more dynamic approach, presented in the next section, that examines the value of the SQL statements executed at run-time might be more suitable for the proposed approach.

4.3.2 Value-Aware DU

A standard state-based DU approach, as described in the previous section, does not take into account the specific values session variables hold nor the specific database statements executed when database invocations are created dynamically.

Algorithm 4 Value-Aware DU: Test Sequence Generation Algorithm for Session Variables - a test sequence is generated for every DU pair for every distinct session variable value.

Require: Test Suite *TS*
Require: Session Variables *SV*
Require: *SV* definition locations *SVDL*
Require: *SV* use locations *SVUL*
Require: *SV* distinct values *Val*

```
1:  TS' = ∅
2:  for all sv in SV do
3:    for all defloc in SVDL do
4:      for all val in Val do
5:        deftest = getdeftestcase(TS,sv,defloc,val)
6:        for all useloc in SVUL do
7:          if usetestid then
8:            usetestid = getusetestcase(TS,sv,useloc)
9:            newSeq = (deftest,usetestid)
10:           TS' = TS' ∪ newSeq
11:         end if
12:       end for
13:     end for
14:   end for
15: end for
16: return TS'
```

Since session variables are used to hold information needed by the server to process the user's next request, it is expected that the value that each variable holds will *ipso facto* affect how the server behaves. For example, a DU pair for a definition of a session variable that holds the user type and a use that checks the user type to perform the appropriate action, would behave differently depending on the value assigned to the session variable in the definition.

Therefore, a dataflow testing approach is developed that takes into account these different values and effectively treats each distinct value as a separate definition. This could be feasible for web applications, even though it is not typically feasible with DU testing of non-web-based applications. This belief is tested empirically in RQ1 of the evaluation.

While performing the dynamic analysis of the original test suite, for each HTTP request the approach also notes session variable values after the request is executed. This information is used together with the static analysis results to identify definition points based on session variable values as well as definition locations.

Algorithm 4, an extension of Algorithm 3, describes the new Value-Aware DU (VADU) testing approach. For every session variable, the algorithm iterates through all definition locations. For every definition location (*defloc*), a test case is retrieved for every distinct value that was observed when dynamically analyzing the original test suite (Line 5). For every use location of the same variable, an HTTP request that covers that location is paired with the definition test case to form a new sequence (Line 9). The output of the algorithm is a set of new test sequences (TS′).

SQL statements can be generated dynamically depending on the user input making it possible for the same database call in the program to execute different SQL statements. Just as different session variable values may have different effects, it is expected that different string values used as SQL calls may also denote different application behaviours.

The approach monitors how each test case interacts with the database by collecting every concrete SQL statement it executes. This information is used to construct the new sequences. Using this dynamic analysis also avoids the limitations of static analyses in the approximation of dynamic strings, since at run-time, the value of the SQL statement string is known.

Algorithm 5 describes the Value-Aware DU approach for generating test sequences for database tables. The algorithm only uses the dynamic analysis results that determine the SQL statements executed by each HTTP request and does not use static analysis. For every database table, all distinct SQL statements (including those generated dynamically) that alter the database are retrieved (Line 4). A test case that executes the definition statement is then paired with all HTTP requests that use the same table. Because *all* uses are appended to the same definition, the computational effort involved in testing multiple uses is reduced. The definition part will only need to be executed once instead of being repeated for every use. However, the algorithm ensures that the path from the definition to every use in the sequence is definition-clear (where no definition-clear path exists, there is no Def-Use pair). Figure 4.2 illustrates how VADU database table sequences are generated.

Algorithm 5 Value-Aware DU: Test Sequence Generation Algorithm for Database - a test sequence is generated for every distinct alter SQL statement and all SQL use statements with distinct paths.

Require: Test Suite *TS*
Require: Table names *Tables*
Require: Distinct SQL alter statements *SA*
Require: Distinct SQL use statements *SU*

```
1:  TS'' = ∅
2:  for all tab in Tables do
3:     newSeq = ∅
4:     for all stmt in SA do
5:        deftest = getdeftestcase(TS,stmt)
6:        newSeq = (newSeq, deftest)
7:        for all stmt in SU do
8:           usetestids = getusetestcases(TS,stmt)
9:           for all testid in usetestids do
10:             newSeq = (newSeq, testid)
11:          end for
12:       end for
13:       if newSeq ≠ deftest then
14:          TS'' = TS'' ∪ newSeq
15:       end if
16:    end for
17: end for
18: return TS''
```

The output of the algorithm is a set of new test sequences (TS″). The union of TS′ from Algorithm 4 and TS″ from Algorithm 5 is the overall output test sequence set for the VADU approach. This approach will be referred to as VADU in the remainder of this chapter.

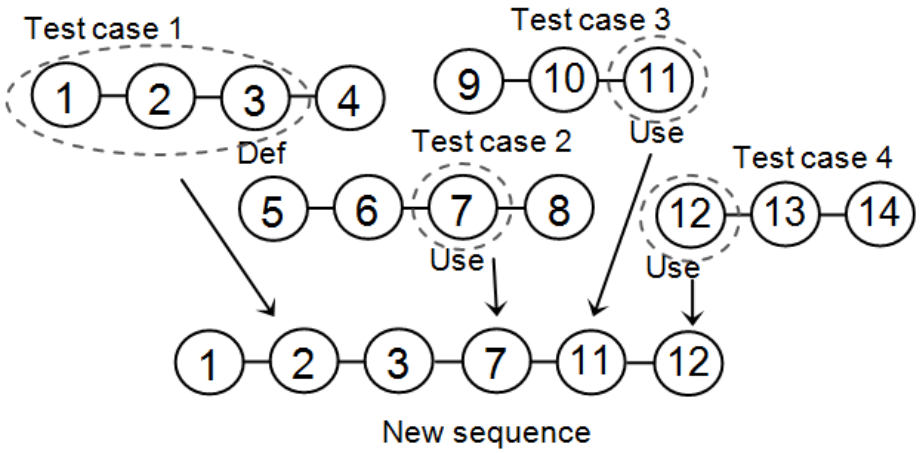


Figure 4.2: VADU Sequence construction technique: The sequence that contains the definition HTTP request is truncated after the request and combined with all use HTTP request to form the new sequence.

4.4 The SART Implementation

Figure 4.3 describes the architecture of the prototype tool SART (State Aware Regeneration Tool) with which the approaches are implemented. The main components of the tool are the Analyser and the Sequence Generator. The Instrumenter instruments the code to record branch coverage. The Test Harness handles the execution of test cases including resetting the database and session state. The Fault Parser implements the oracle described in Chapter 1.5.2.

The Analyser consists of a static and dynamic component. The output of the Analyser is saved in the Analysis Data repository and later used by the Sequence Generator.

The static component determines session variable Def-Use locations from the source code. The static component is written in Stratego/xt [BKVV08] and PHP-Front [BB10].

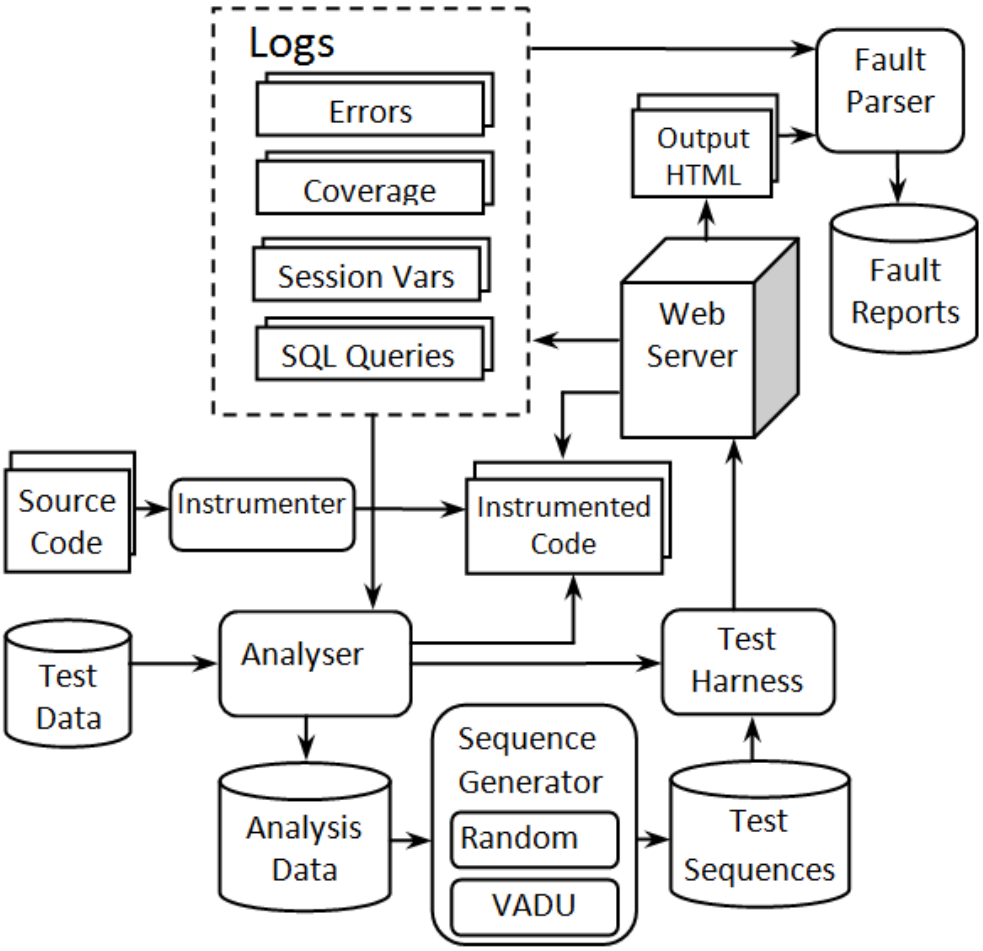


Figure 4.3: SART architecture

The dynamic component executes the test cases in the original test suite using the Test Harness. When executing the test cases, the server logs session variable values to the Session Vars log, executed database queries to the SQL Query log and statement coverage and errors to the Coverage and Error logs. The Analyser parses the logs to analyse each test case and saves the results in the Analysis Data repository.

To determine which test cases execute the statements that define or use the state, SART uses Xdebug [Ret12] to record statement coverage in the Coverage log for each test case and then matches the result with session definition and use locations.

To determine session variable values for each HTTP request, SART implements a function that records the session variables' values in the Session Vars log for subsequent access by the Analyser. SART uses the PHP configuration settings to prepend the file that defines the function and registers it as an exit function that is automatically called at the end of each request.

To analyse which test cases alter the database or use it, the SQL server is configured to log every query to the SQL Query log which is then parsed by the Analyser.

The Sequence Generator implements Algorithms 4 and 5 and uses the Analysis Data to generate new test sequences. The new sequences are then executed by the Test Harness and coverage and fault data is measured.

The Sequence Generator and Test Harness are both implemented in Perl and use the HTTP, HTML and LWP (Library for WWW in Perl) libraries.

4.5 Evaluation

The evaluation is designed to answer the following research questions:

RQ1: How many new sequences are generated using VADU compared to the original test suite?

This question investigates the feasibility of the proposed VADU approach. The number of test sequences generated for a standard State-based DU approach depends on the number of definitions and uses present in the code. However, the Value-Aware DU (VADU) approach defined in Section 4.3.2 generates a different sequence for each value. Therefore, it might generate such a large number of test sequences that it becomes infeasible. RQ1 investigates whether this explosion in VADU test cases occurs in practice for the four web applications studied. In answering RQ1, the results of the static analysis are also examined and the numbers of session variables and database calls for each application are reported.

RQ2: How much can branch coverage be improved?

This question investigates the effectiveness of VADU in terms of branch coverage. The proposed approach generates test sequences that define and use the state in ways not present in the original test suite. This is expected to enhance branch coverage of the regenerated test suite compared to the original test suite from which it was constructed. Additional branch coverage is measured and results reported for VADU.

RQ3: How much can fault finding ability be improved?

This question investigates the effectiveness of the approach; as with other testing approaches, the real test is fault detection, not merely coverage.

For RQ2 and RQ3 the evaluation also compares VADU to random generation of sequences, to provide a baseline for comparison. Random sequences are used as a baseline for comparison because random construction of request sequences has previously been proposed in the literature [ERKI05, SGSP05b] and so it denotes the current state-of-the-art for web application test suite regeneration.

4.5.1 Experimental Set-up

The evaluation is performed on four of the six applications described in Section 1.5.1. These four applications use both a database and session variables and are therefore good candidates for the evaluation. PHPSysInfo and PHPBB2 were excluded because both applications do not use session variables and PHPSysInfo does not use a database.

To construct starting test suites for the evaluation, 30 test suites were sampled from test cases generated by SWAT (Chapter 3). All the test suites are chosen to have branch coverage within 10% of the maximum coverage that can be achieved by SWAT. Test suites with relatively high coverage are chosen, when possible, because increasing coverage for already high-coverage suites is expected to be harder than seeking to improve low coverage suites. That is, test suites were chosen to which any increase in coverage or fault detection would be non-trivial to obtain, in order to pose a demanding challenge to the evaluated approaches.

The experiment applies the Value-Aware DU approach (VADU) defined in Algorithms 4 and 5 and random regeneration to the sampled 30 test suites of the four applications. The improvements in branch coverage and faults found in relation to the original test suite are measured and compared. The computational cost that was needed to achieve these improvements is also measured and compared for VADU and the random regeneration approach.

Computational cost is measured as the total number of HTTP requests that needed to be executed to achieve the final improvements in coverage and faults. The number of requests executed can be viewed as a better measure of computational cost because it is not affected by the specifics of machine and platform on which experiments are performed. However, total execution time is also measured and reported in order to provide data on realistic performance expectations. The evaluation was performed on an Intel Core i5-2450M CPU, running at 2.50 GHz with 2 GB RAM.

For random sequence generation, an algorithm that will be simply called 'Random' hereafter, the same number of test sequences reported for VADU is generated and coverage and faults found are measured. The implementation for Random ensures that only distinct sequences are generated. The way two HTTP requests are combined for Random follows a similar principle to that depicted in Figure 4.1: When an HTTP request is chosen to form the first part of the new sequence, any leading requests that set up the state are also included.

The automated oracle described in Section 1.5.2 is used to identify faults revealed by the generated new sequences.

A Wilcoxon paired one-sided signed rank test at the 95% confidence level is performed to determine the statistical significance of the observed results. The Wilcoxon test is used because it is non-parametric and the evaluation wishes to make assumptions about neither the distribution of coverage values nor the faults found. The test is paired because each of the 30 runs of each of the two algorithms starts with the same initial test suite. The test is one-sided because it is known from the results that the median of VADU is above that of Random.

4.5.2 Results

In this section, the results obtained from the evaluation on the four web applications for each approach are presented.

Analysis Results and Number of Generated Sequences

Table 4.1 reports the information obtained from the static analysis of the four applications: The number of session variables and database tables together with their definitions and uses. We notice that for Timeclock the static analysis was not able to extract all definitions and uses of database tables. When investigating the reason, it was discovered that when SQL statements are formed in Timeclock, table names are constructed dynamically by reading a prefix that has been set at installation time from the database. Since table names are constructed dynamically, static analysis is unable to discover their definitions and uses. This observation provides a further justification for the advocacy of a dynamic approach when generating sequences for database tables that is provided in VADU.

An analysis of the session variables found for each application reveals that for three applications, a relatively low number of session variables (2-3 variables) were discovered and these variables were used to keep track of the logged-in users. Webchess also has other session variables that hold information about the selected game and preferences. The analysis also shows that session functions (discussed in Section 4.3.1) that use the state are not used frequently (only in FaqForge).

Table 4.1: Static analysis results: numbers of session variables, functions, tables and Def-Use locations.

	Sessions						Database		
	Variables			Functions			Tables		
App Name	num	Defs	Uses	num	Defs	Uses	num	Defs	Uses
FAQForge	2	2	2	7	5	4	2	11	22
Schoolmate	3	3	6	2	2	0	15	79	215
Webchess	11	28	112	2	7	0	6	38	55
Timeclock	3	3	68	2	34	0	3	1	2

Table 4.2: Average numbers of generated test sequences (seqs) and requests for VADU for both session variables and database tables for 30 test suites for each of the four applications.

	Original		Sessions		Database		Total	
App Name	Seqs	Requests	Seqs	Requests	Seqs	Requests	Seqs	Requests
FAQForge	34	72	7.6	15.1	9.1	145.5	16.7	160.6
Schoolmate	174	368	31.9	63.7	83.4	3,706.0	115.3	3769.7
Webchess	44	95	109.7	219.4	4.4	68.8	114.1	288.2
Timeclock	244	507	133.3	266.5	2.4	102.4	135.7	368.9
All apps	123	260	70.6	141.2	24.8	1,005.7	95.4	1146.9

Table 4.2 shows the number of sequences generated for VADU for both session variables and database tables. Random was not included because the number of sequences generated for Random is the same as VADU. The table also reports the number of test cases in the original test suite. The increase in the average number of sequences

VADU adds to the original test suite over all applications is 77.6% (95.4 sequences added to the original 123 sequences). The increase in the average number of total requests in those sequences is 4.4 times the average number of requests in the original test suite. However, a closer look at the results reveals that the highest increase in the number of generated requests comes from database table requests in Schoolmate. This could be caused by the fact that Schoolmate has the highest number of database transactions in the original test suites. This suggests that an imporvement to the algorithm is needed to minimize the number of requests generated by analysing the values of SQL statements to select a subset that is more likely to be effective when generating sequences. However, as the analysis of computational cost reveals, this increase in test cases is comfortably manageable with reasonable time bounds.

One interesting observation is that several test cases that execute a definition or use point for database tables have been discarded when generating sequences. By using values not definition points, VADU can be more precise, eliminating false definition points. That is, when using VADU, SQL statements are collected and parsed dynamically, making it possible to exclude SQL statements that are invalid (and would be rejected by the database server). These invalid SQL statements are excluded because they would have no effect on the database and therefore are neither definitions nor uses. These invalid SQL statements are created by dynamic generation of SQL statements that contain user inputs. If these user inputs are not validated before being concatenated to SQL statement fragments, the final statement may be invalid.

Coverage

Branch coverage results are reported in Table 4.3. The reported results are calculated as the improvement in percentage over coverage of the original test suites. Each experiment is repeated 30 times to allow for statistical significance testing and to cater for variations in algorithm performance for different starting test suites. Therefore, the mean and median coverage (and fault detection) values are reported in the table.

The results indicate that VADU performs better than Random for all four applications. This is a particularly pronounced effect for Webchess where the mean additional improvement in coverage over Random is 16.17% (25.31% for VADU compared to 9.14% for Random). The difference in median is even higher (33.18% for VADU compared to 3.21% for Random).

Table 4.3: Test case generation results: Improvements in coverage and faults found are calculated in relation to the original test suite. For improvements, values in bold are statistically significantly better than values above them using Wilcoxon paired one-sided signed rank test at the 95% confidence level.

	Original					% Improvement				Computational	
	Coverage		Faults			Coverage		Faults		Cost	
App Name	mean	median	mean	median	Algorithm	mean	median	mean	median	Requests	Time (sec)
FAQForge	67.49	67.61	50.97	50.00	Random	2.71	2.13	0.00	0.00	223	34
					VADU	**14.10**	**14.21**	4.12	4.00	291	53
Schoolmate	66.32	66.30	96.30	95.50	Random	1.79	1.49	2.85	3.03	4,115	731
					VADU	**14.42**	**14.34**	**14.31**	**13.66**	4,547	781
Webchess	38.20	38.06	67.83	68.00	Random	9.14	3.21	1.78	1.44	369	151
					VADU	**25.31**	**33.18**	**9.30**	**8.82**	537	273
Timeclock	18.11	18.12	177.40	178.00	Random	0.15	0.15	0.07	0.00	823	381
					VADU	**5.11**	**5.10**	**9.02**	**8.99**	1,366	615
All apps	47.53	52.60	98.13	74.50	Random	3.45	1.60	1.18	0.00	1,383	317
					VADU	**14.74**	**13.68**	**9.19**	**8.97**	1,685	431

An investigation of the causes of this strong performance found that in Webchess, a session variable **gameID** is used to store the game selected by the player. If the selected game is valid and active, pairing the test case that selects it with other HTTP requests that perform different actions to cover DU paths greatly increases coverage. These findings confirm the usefulness of the VADU approach since it is sensitive to the values stored in session variables and would therefore ensure that all DU pairs are exciersied for every value.

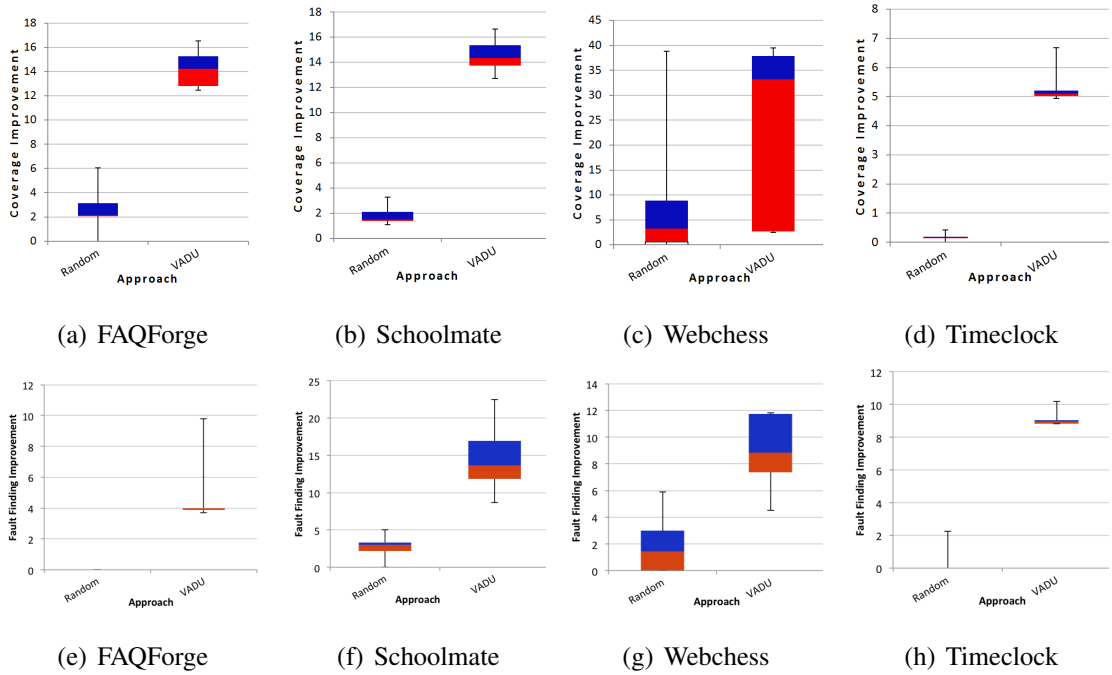
The top row of box plots of Figure 4.4 demonstrates the variations in coverage improvement over 30 test suites for each of the four applications for each approach. For Webchess (Figure 4.4(c)), the improvement in coverage for VADU can be as small as 1.3% and as high as 43% (over the 30 trails). When this peculiarly high variance was examined, it was observed that the **gameID** session variable also played a pivotal role in these observations. The coverage improvement is limited in cases where the original test suite fails to include a single test case that selects a valid and active game. This suggests a relationship between the quality of the original test suite and the effectiveness of the approach. In this case, it also suggests a potential fault because the application allows the user to select invalid values for **gameID** and registers these values in the session variable without checking the validity of the selected game.

Faults

Fault detection results for the 30 original test suites and the new test sequences generated by VADU and Random for the four applications studied are reported in Table 4.3.

The Random algorithm does not find any new faults in FaqForge and only finds new faults for Timeclock in one of the 30 trails. The overall mean of improvements in fault detection for Random is 1.18%. VADU finds an overall mean of 9.19% new faults that were not discovered by the original test suite with a median of 8.97%.

Although Random improves branch coverage for FaqForge and Timeclock, the results of the evaluation show that it is not as effective at fault detection. This suggests (as is widely believed for non-web applications also) that although coverage affects fault finding ability, other factors also influence the effectiveness of a test suite in finding faults.



(a) FAQForge (b) Schoolmate (c) Webchess (d) Timeclock

(e) FAQForge (f) Schoolmate (g) Webchess (h) Timeclock

Figure 4.4: Variations in coverage and fault detection improvement results over 30 test suites for VADU and Random on each of the four web applications. The top row illustrates branch coverage improvements while the bottom row shows fault detection. The y-axis is the improvement(%) in branch coverage (or faults found) compared to the coverage (or faults found) for the original test suites.

The bottom row of box plots in Figure 4.4 shows the variations in the percentage improvement in faults found for the two approaches over 30 test suites for each of the four applications compared to the original test suites. It is interesting to observe that, for Webchess, although there is an overlap in the performance of VADU and Random for branch coverage, in fault finding, this overlap is much smaller. This suggests that the way two sets of test sequences are constructed (in this case VADU and Random) has an effect on fault detection even when coverage is comparable.

Computational Cost

In this section, the computational cost of the experiments are reported (executed requests and elapsed time). However, in this evaluation all processes are fully automated including checking the oracle and reporting fault results. Since the whole process is automated, differences in elapsed time merely mean that a tester needs to wait a few minutes longer for the results (VADU, the most computationally expensive algorithm,

takes an average of 13 minutes for the slowest application).

Results for the computational cost spent to achieve the reported improvements in branch coverage and fault detection are presented in Table 4.3. Computational cost is represented using two measures: Number of requests executed and total execution time. The number of requests is the total number of HTTP requests that needed to be executed to achieve the reported improvements. This includes every HTTP request that was executed for the dynamic analysis needed for VADU as well as the execution of the new sequences and the measurement of improvements.

Naturally, VADU requires more requests than Random since Random does not require dynamic analysis. Execution time is measured as the total elapsed time for all activities needed to generate and execute the new sequences. Random is faster than VADU running , on average, in 5.28 minutes (317 seconds) over all applications with the slowest application (Schoolmate) running, on average, in 12.18 minutes (731 seconds). VADU, on average, takes 7.18 minutes (431 seconds) over all applications. The slowest application is also Schoolmate, taking, on average, 13 minutes (781 seconds).

These results show that with the improvements in coverage and faults found (reported in previous sections) the overhead in execution times is relatively small, and certainly within acceptable bounds even on standard equipment.

4.5.3 Answers to Research Questions

In this section, the research questions posed at the beginning of the evaluation section are answered, based on the empirical results from the evaluation.

RQ1: How many new sequences are generated using VADU compared to the original test suite?

The empirical evaluation showed that the increase in the number of requests generated for the combination of session variables and database tables is, on average, 4.4 times the average number of requests in the original test suites. However, for three of the four applications the increase was, on average, 1.2 times; this suggests that VADU is feasible. This also suggests that further improvements that minimize the number of sequences/requests generated is needed.

RQ2: How much can coverage be improved?

VADU improves branch coverage for all four applications studied compared to the original test suite by a mean of 14.74% and performs better than Random for all four applications.

Statistical testing confirms that these improvements, for VADU compared to Random, are statistically significant for all four applications using Wilcoxon paired one-sided signed rank test at the 95% confidence level.

RQ3: How much can fault finding ability be improved?

The empirical evaluation showed that using the automated oracle, VADU can increase the number of faults found by a mean of 9.19%, compared to the original test suite (over the 30 test suites evaluated for the four web applications). VADU performs better than Random over all test suites and all four applications. The Wilcoxon paired one-sided signed rank test at the 95% confidence level confirms that these improvements are statistically significant.

4.5.4 Threats to Validity and Limitations

Internal threats: The internal threats that affect the validity of the results depend on the implementation of the approaches and the set-up of the evaluation. To minimize factors that affect the measurement of execution times, code was shared between the implementations of the approaches whenever possible. The number of requests executed is also provided as an additional measure of computational cost, this is an algorithmic rather than implementation specific quantity.

External threats: The applications that were chosen and the starting test suites used might affect the degree to which the results can be generalized. The applications chosen were used by previous research on web application testing and are also used in current practice. These applications were selected also because they use both session variables and a database. The test suites selected can also affect results. This is why 30 different test suites were sampled for each application that have relatively high coverage. The source of the starting test suites (SWAT) might also affect the ability to generalize results. Therefore, further experiments using test suites from other sources are needed to confirm the reported results.

Construct threats: Construct threats are related to the measurements used to compare the three approaches. Branch coverage and faults found were used, two measurements that are widely used in research to compare effectiveness.

Limitations: The current implementation of the tool is not able to handle complex SQL statements that are, for example, nested or use joins. The tool will only be able to recognize the first table that is used. Enhancing the tool to handle these SQL statements types may further improve effectiveness. The implementation generates test sequences to cover DU pairs without checking whether the original test suite already contains a test case that covers the pair. To enhance the implementation to check the original test suites might reduce the number of sequences produced and make the approaches still more efficient.

4.6 Related Work

Liu et al. [LKHH00a, LKHH00b] used traditional dataflow modelling of the server code, seeking to generate test cases for structural coverage of web application server-side code (page 67 of the literature review). By contrast, the approach proposed in this chapter seeks to generate test sequences that take into account client interactions (using browser functions) and targets the server-side state (including session variables and database tables), whereas Liu et al. concentrate on server-side source code variables (applying traditional Def-Use testing in much the way in which it was originally designed for non web-based applications).

The argument of this chapter is that the interactions of state have a pivotal role in affecting the application's behaviour, and require a different approach to dataflow testing. Simply testing server-side code for structural coverage may overlook the effect of these interactions.

In session based testing (Section 2.3.2), several random regeneration experiments were carried out. Sprenkle et al. [SGSP05b] investigated the effect of state on coverage and fault finding by executing sessions in different random orders without initializing the state. The experiment showed that different request orderings result in elevated coverage and the detection of new faults. Elbaum et al. [EKR03, ERKI05] also suggested combining parts of different sessions to produce new test suites. These previous studies indicated that recombining or reordering test cases can lead to more effective test

suites. However, the new test suites were generated purely randomly. By contrast, this chapter proposes an approach to produce these new test suites by analyzing the effect on the server-side state. As the evaluation results showed, this state-aware approach significantly improves both server-side coverage and fault detection.

As discussed on page 42 of the literature review, Alshahwan and Harman [AH08] repaired test sequences for regression testing by adding and/or removing requests from sequences. The repairs where based on changes in how the application is connected (by links), while this chapter considers state interactions to regenerate request sequences that improve coverage and fault detection.

Several test data generation approaches have been proposed for web applications in the literature [AH08, AH11, AKD+08, AKD+10, HAO09, HO08, RT01a, WYC+08]. The test suites generated by these approaches aim to maximize structural coverage and all are thus good candidates for the production of the starting test suite required by the proposed regeneration approach.

Regeneration and augmentation has also been applied to test suites for conventional applications, [Arc10, SCA+08, YH10], but these approaches have not, hitherto, been applied to web applications, with their separation of client-side and server-side and their close coupling to back-end databases.

Other authors have also considered the problem of testing database applications [CC99, CDF+04, DFW04, EMS07, HO06], though these focus on structural coverage of database dependent branches, whereas the approach proposed in this chapter targets database state interactions for web-based applications.

4.7 Conclusion

This chapter introduced an approach to regenerate test sequences from existing pools of HTTP requests present in the original test suites. The approach proposed exploits server-side state manipulation to generate new test sequences that define and use the state in ways not present in the original test suite. This chapter introduced the Value-Aware DU approach that is aware of the values and specific state modifying SQL statements as well as traditional DU pair information.

A tool, SART (State Aware Regeneration Tool), was introduced that implements the proposed approach for PHP applications. The results of an empirical evaluation on

four real world web applications are also reported. The evaluation reports and compares branch coverage information and faults found for the approach to the random recombination approach currently advocated in the literature. The obtained results provide evidence to support the claim that the novel value aware approach significantly increases coverage and fault detection (at reasonable computational cost).

Chapter 5

Output Uniqueness Criteria

5.1 Introduction

In Chapters 3 and 4 the output was used to enhance the generation of test data and test sequences. Two tools, SWAT and SART, were introduced with the aim of maximizing branch and Def-Use coverage to produce test suites that are effective in finding faults. The evaluations in the previous chapters indicated that coverage is not the only factor that influences fault finding. For example, although Random and VADU achieved similar coverage levels for Webchess for some of the test suites used in the evaluation, VADU was more effective at fault detection. This finding replicates similar findings is previous testing research [NA09, HFGO94]. This chapter proposes using the output as a test adequacy and selection criterion.

Structural coverage remains prevalent as the only adequacy criterion considered in many studies of software testing. This chapter argues that we need more research on complementary criteria and proposes one based on output uniqueness. Although higher coverage may increase fault detection, there remains much controversy about coverage being the only contributing factor [NA09, HFGO94]. This chapter proposes a novel criterion that is based on the uniqueness of the program's output to enhance traditional coverage criteria. This approach expects that raising the diversity of the output could lead to test suites that are more effective at exposing faults.

Faults with high severity often propagate to the observable output and affect user perception of an application [DW10]. Therefore, a system's output may provide a valuable resource for identifying unexpected behaviour that is considered more critical from a user's point of view. In applications with rich output structure, such as web applica-

tions, output may prove to be particularly suited as a new criterion. The complexity and richness of the output may also make it more likely for faults to propagate to the output and for the approach to be effective. This is the proposition which this chapter investigates.

This chapter introduces a new output uniqueness criterion and provides seven definitions of unique output applied to web applications. It investigates the use of those output uniqueness definitions both as complimentary test selection criteria to structural coverage and as alternative criteria when structural coverage cannot be measured. First, the chapter introduces an approach that applies the proposed uniqueness criteria to augment test suites that were created using a traditional structural coverage criterion (branch coverage) with test cases that provide unique outputs. Finally, an empirical study is provided that investigates the correlation between test suite effectiveness in fault detection, structural coverage and the new output uniqueness criteria.

The specific contributions of this chapter are as follows:

1. The introduction of a novel test adequacy criterion based on output uniqueness.

2. Seven definitions of output uniqueness in the context of web applications.

3. An initial evaluation on five real world web applications of four of the output uniqueness definitions in terms of their ability to find real faults. The initial results indicate that output uniqueness is a valuable criterion for generating test suites that are more effective at exposing faults. Uniqueness outperforms random augmentation by an average of 280% over all applications.

4. An empirical study for six web applications that investigates the effectiveness of output uniqueness criteria and their correlation with both fault detection and structural coverage.

The rest of this chapter is organized as follows: Section 5.2 discusses the output of web applications and presents the new output uniqueness definitions. Section 5.3 presents the initial experiment for test suite augmentation using output uniqueness criteria. Section 5.4 presents the empirical evaluation of output uniqueness criteria together with a discussion of the results. Section 5.5 presents related work and Section 5.6 concludes.

5.2 Output Uniqueness Criteria

Checking the output of a test case is a comparatively cheap operation: It neither requires instrumentation of the code nor the building of intermediate structures, such as control flow graphs or dataflow models. This makes it comparatively easy to experiment with a large number of test cases to find the ones that provide interesting or 'unique' outputs. Checking the output also does not require access to the source code of the application making it possible to generate potentially effective test data for components where the source code is unavailable, such as third party components. This section discusses web application client-side output and proposes seven definitions of output uniqueness to be used in test case selection.

Web Application Output

The principal output of a web application, visible to the user, is a client-side HTML page. The client-side page can be either static or dynamically generated. This page is composed of the content, the HTML structure and embedded elements, such as images or client-side scripts (e.g., JavaScript). The approach proposed in this chapter focuses on the content and HTML structure of the client-side page:

The Content: The content (C) is the textual data that is presented to the user. The main element in this content is usually the data that the user requested. For example in case of a search, the content is the list of matching items. Extra helping text can also be found throughout the output page. Examples of such text are page titles, welcome messages and field labels.

HTML Structure: The HTML structure (H) defines how the page is presented to the user. The content is organized in tables or frames that are constructed using HTML code. Aesthetic elements, such as colours and fonts as well as backgrounds and embedded pictures can also be specified using HTML code. In addition to the page's appearance, HTML can be used to define functional elements that enable the user to interact with the application. HTML `forms` and links are primary examples of these elements. HTML code is constructed using HTML tags (T) that define the type of the element (e.g., table, form, input). Each tag has a number of relevant attributes (A), such as actions for `forms` or values for input.

A web application's client-side page output can be defined as a tuple $O{=}{<}C,H{>}$ where the HTML code H can be defined as a set of tags T and each tag in T consists of a set of attributes A.

Client-side Page Output Uniqueness

To use the proposed output based test selection criteria, first 'uniqueness' needs to be defined. A strict definition of output uniqueness could capture all test cases that cause a fault that propagates to the output. However, a strict definition could also lead to an explosion in the test suite size, with many additional test cases that may yield no additional benefit.

The aim of the proposed approach is to evaluate a number of output uniqueness definitions to identify the most effective definition that captures interesting output differences while maintaining a smaller test suite.

The HTML code in Figure 5.1 is taken from one of the applications studied (Schoolmate) and simplified for readability. This code will be used to demonstrate which parts of the output are considered for each proposed definition.

A test suite can be defined as a set of (input, output) pairs. The strictest definition to consider is:

Definition Output o is OU-All unique with regard to a test suite $T \iff$ for all (i, o') there exists at least one observable difference between o and o'.

When a new output page is analysed, any difference in any element of the page compared to all previously visited pages categorizes the new output page as unique. All the HTML code in Figure 5.1 will be considered for comparison when using OU-All.

This definition could potentially lead, in some cases, to infinitely many unique outputs that do not necessarily enhance the test suite's effectiveness, but considerably increase the oracle cost. For example, an application that displays the date on the output page could result in a potentially infinite set of unique outputs. A page that displays product information would have as many unique outputs as there are products in its database. To overcome this problem output uniqueness can be defined, less strictly, in terms of the HTML structure of the page ignoring the text.

Definition Output o is OU-Struct unique with regard to a test suite $T \iff$ for all (i, o') where $o =<c, h>$ and $o' =<c', h'>$ there exists at least one observable difference between h and h'.

Figure 5.2 shows the part of the output page that will be considered for comparison when using OU-Struct. The text in the output page is removed and only the HTML

```
<html>
 <head>
 <title>SchoolMate - School Name</title>
 </head>
  <body>
 <form action='./index.php' method='post' name='login'>
 <table width='100%' height='85%' align='center' >
  <tr>
   <td align='right'><b>Username:</b></td>
   <td><input type=text name='username'></td>
  </tr>
  <tr>
   <td align='right'><b>Password:</b></td>
   <td><input type=password name='password'></td>
  </tr>
  <tr>
   <td align='center'>
   <input type=submit value='Login' ></td>
  </tr> </table>
   <input type='hidden' name='page' value='1'>
   <input type='hidden' name='login'>
  </form>
  <span class='footer'>Powered By -SchoolMate</a>
 </body>
 </html>
```

Figure 5.1: Simplified example HTML taken from Schoolmate to demonstrate which part of the output is used for each output uniqueness definition.

structure is retained and compared to previously observed output to decide if the output is new.

This definition eliminates the 'potentially infinite output' issue in the text discussed for OU-All. However, the HTML structure may still yield large test suites. Consider the product pages of items again, if the `form` to order an item contains a hidden field that holds the item's ID, there will be as many unique outputs as there are products in the database. A new definition of output uniqueness can be proposed where the HTML structure of a page is stripped of any text or embedded values and only the opening and closing tags are considered. This is to eliminate any variations caused by `form` options, default values or font and style settings.

```
<html>
 <head>
 <title></title>
 </head>
  <body>
 <form action='./index.php' method='post' name='login'>
 <table width='100%' height='85%' align='center' >
  <tr>
   <td align='right'><b></b></td>
   <td><input type=text name='username'></td>
  </tr>
  <tr>
   <td align='right'><b></b></td>
   <td><input type=password name='password'></td>
  </tr>
  <tr>
   <td align='center'>
   <input type=submit value='Login' ></td>
  </tr> </table>
   <input type='hidden' name='page' value='1'>
   <input type='hidden' name='login'>
  </form>
  <span class='footer'></a>
 </body>
 </html>
```

Figure 5.2: The part of the output considered for OU-Struct: The text in the output page is removed and only the HTML structure is used to decide if the output is new.

Definition Output o is OU-Seq unique with regard to a test suite $T \iff$ for all (i, o') where $o =<c, h>$ and $o' =<c', h'>$ and where h and h' contain a set of tags t and t' and attributes a and a' respectively and there exists at least one observable difference between t and t'.

Figure 5.3 shows the part of the output that will be considered for OU-Seq. In addition to removing all text, all attributes from HTML tags are removed.

The previous two definitions focused on the HTML structure of a page. However, the text in the page can contain error messages produced by the server. Therefore, another definition of output uniqueness is added:

```
<html>
 <head>
 <title></title>
 </head>
  <body>
 <form>
 <table>
  <tr>
   <td><b></b></td>
   <td><input></td>
  </tr>
  <tr>
   <td><b></b></td>
   <td><input></td>
  </tr>
  <tr>
   <td>
   <input></td>
  </tr> </table>
   <input>
   <input>
  </form>
  <span></a>
 </body>
 </html>
```

Figure 5.3: The part of the output considered for OU-Seq when deciding if an observed output is new. All text is removed from the output page as well as all attributes in HTML tags.

Definition Output o is OU-Text unique with regard to a test suite $T \iff$ for all (i, o') where $o = <c, h>$ and $o' = <c', h'>$ there exists at least one observable difference between c and c'.

Figure 5.4 shows the part of the output that will be considered when using OU-Text. The HTML structure is removed, leaving only the text in the output page.

The previous four output uniqueness definitions considered the client-side page as a whole (OU-All) and decomposed it to its HTML (OU-Struct) and text (OU-Text) elements and also stripped the page's HTML to its basic elements (OU-Seq). However, the values of specific parts of both the text and HTML could indicate interesting exe-

```
SchoolMate - School Name
      Username:
      Password:
Powered By -SchoolMate
```

Figure 5.4: The part of the output considered for OU-Text when deciding if an observed output is new. Only the text in the output page is retained for comparison.

cutions of the application under test. The next three definitions focus on those specific parts of the output page.

Definition Output o is OU-Hidden unique with regard to a test suite $T \iff$ for all (i, o') where $o =<c, h>$ and $o' =<c', h'>$ and h and h' contain a set of tags t and t' and attributes a and a' respectively there exists at least one observable difference between t and t' or between a_{name} and a'_{name} or a_{hidden} and a'_{hidden}.

The subscripts to an attribute a denote the type of attribute being considered. An attribute a_{name} is an attribute of type name while a_{hidden} is an attribute that describes the value of a hidden field.

This definition is similar to the OU-Seq definition but also takes into account hidden `form` variables and their values. Hidden `form` variables are control variables that are embedded by the server-side code in `form`s in the output HTML to pass state over subsequent requests. Therefore, they are expected to be significant in describing how the application behaved on a previous request. Unexpected values held in hidden `form` variables can possibly indicate a fault in a previous execution. Figure 5.5 shows the part of the output that will be considered for OU-Hidden.

However, as discussed before, hidden `form` variables can lead to a uniqueness definition identifying infinitely many unique outputs depending on the design of the application and how hidden `form` fields are used within it. Therefore, it might be useful to consider characteristics of hidden `form` variable values rather than their actual values. A new definition is proposed that is based on the subtypes of hidden `form` variable values. These considered subtypes are: positive and negative numbers, strings, zeros, empty strings and `NULL`. These subtypes are chosen to be general and not application specific for the experiments performed in this section to avoid bias and keep

```
<html>
 <head>
 <title></title>
 </head>
  <body>
 <form>
 <table>
  <tr>
   <td><b></b></td>
   <td><input name='username'></td>
  </tr>
  <tr>
   <td><b></b></td>
   <td><input name='password'></td>
  </tr>
  <tr>
   <td>
   <input value='Login'></td>
  </tr> </table>
   <input  name='page' value='1'>
   <input  name='login'>
  </form>
  <span></a>
 </body>
 </html>
```

Figure 5.5: The part of the output considered for OU-Hidden when deciding if an observed output is new. All text and attributes are removed except for attributes that describe a tag's name or a hidden value.

the evaluation generalized. However, it might be useful in practice to define subtypes specifically for an application (or even specific subtypes for each hidden input) based on the tester's domain and application knowledge.

Definition Output o is OU-Subtypes unique with regard to a test suite $T \iff$ for all (i, o') where $o =<c, h>$ and $o' =<c', h'>$ and h and h' contain a set of tags t and t' and attributes a and a' respectively and there exists at least one observable difference between t and t' or between a_{name} and a'_{name} or between the subtypes of a_{hidden} and a'_{hidden}.

```
<html>
 <head>
 <title></title>
 </head>
  <body>
 <form>
 <table>
  <tr>
   <td><b></b></td>
   <td><input name='username'></td>
  </tr>
  <tr>
   <td><b></b></td>
   <td><input name='password'></td>
  </tr>
  <tr>
   <td>
   <input value='Login'></td>
  </tr> </table>
   <input  name='page' value=num>
   <input  name='login' value=NULL>
  </form>
  <span></a>
 </body>
 </html>
```

Figure 5.6: The part of the output considered for OU-Subtypes when deciding if an observed output is new. All text and attributes are removed except for attributes that describe a tag's name or a hidden value. The values of hidden values are replaced by predefined subtypes.

Figure 5.6 shows the part of the output that will be considered for OU-Subtypes. The value of the first hidden field `page` was replaced by the corresponding subtype 'num' and because `login` has no value the subtype attached to it is `NULL`.

Finally, a new definition is introduced for the text element of the output. If an error occurs during execution, this might cause the execution to terminate and an error message to be printed. This error message would be printed on the last line of the output before termination therefore the last line of text in the output might prove to be a useful output uniqueness criterion.

Definition Output o is OU-LastText unique with regard to a test suite $T \iff$ for all (i, o') where $o =<c, h>$ and $o' =<c', h'>$ there exists at least one observable difference between the last line of c and c'.

For the HTML example in Figure 5.1, only the last line `Powered By - SchoolMate` will be considered for comparison.

5.3 Augmenting Test Suites Effectiveness by Increasing Output Diversity

This section investigates using output uniqueness in test suite augmentation by adding test cases with unique outputs not present in the original test suite. Section 5.3.1 presents the augmentation approach whilst Section 5.3.2 evaluates the proposed approach on test suites generated by SWAT (Chapter 3) for five web applications.

5.3.1 Approach

This section proposes that output uniqueness should be used to augment a traditional test generation adequacy criterion not replace it; output alone would not capture all faults that can manifest themselves in an application. For example, two different errors that cause a web application to return a blank page would have the same output. However, as the evaluation in Section 5.3.2 shall show, focusing on output uniqueness improves fault finding ability, even for relatively high coverage test suites.

This section uses output uniqueness to augment a test suite generated to satisfy a traditional test adequacy criterion (branch coverage) and introduces an approach to use the first four definitions presented in Section 5.2 in test suite augmentation.

Algorithm 6 describes the augmentation approach which uses output uniqueness to generate new test cases. The algorithm takes a test suite as an input and builds new test suites that satisfy each of the four basic output uniqueness definitions in Section 5.2. For each original test case, one input is mutated at a time to generate a new test case. The input is mutated, with equal probability, by either assigning a random value or a value collected dynamically from the output of the original test suite. The new mutated test case is then executed and the output is examined to determine which output uniqueness definition it satisfies. For each definition a test suite is maintained that contains all mutated test cases that satisfy its definition.

5.3.2 Evaluation

The evaluation is designed to answer the following two research questions:

RQ1: Does using output-uniqueness augmented test suites enhance fault finding ability?

RQ2: How do the four definitions of output uniqueness affect the fault finding ability and test effort of the generated test suite?

To answer these two questions a test suite generated using SWAT from Chapter 3 is augmented with test cases that enhance output uniqueness for each of the four definitions. The fault finding ability of the original suite is then compared to the augmented suites and also to the original SWAT test suite augmented by the same number of additional test cases selected randomly. The number of faults per new test case of each of the definitions compared to random is then calculated and compared. Wilcoxon paired one-sided signed rank test at the 95% confidence level is performed to determine the

Algorithm 6 Test Data Generation Algorithm: Starting from a test suite the algorithm generates four test suites that each satisfy one of the output uniqueness definitions

Require: Test Suite *TS*
1: **for all** T in *TS* **do**
2: *output* = executeTestCase(T)
3: $\mathcal{O} = \mathcal{O} \cup$ *output*
4: **end for**
5: **for all** T in *TS* **do**
6: **while** Number of tries $<= 100$ **do**
7: $T' =$ mutateInput(T)
8: *output* = executeTestCase(T')
9: **if** OU-AllSatisfied($\mathcal{O}$,*output*) **then**
10: *TS-ALL* = *TS-ALL* $\cup$ T'
11: **if** OU-TextSatisfied($\mathcal{O}$,*output*) **then**
12: *TS-Text* = *TS-Text* $\cup$ T'
13: **end if**
14: **if** OU-StructSatisfied($\mathcal{O}$,*output*) **then**
15: *TS-Struct* = *TS-Struct* $\cup$ T'
16: **end if**
17: **if** OU-SeqSatisfied($\mathcal{O}$,*output*) **then**
18: *TS-Seq* = *TS-Seq* $\cup$ T'
19: **end if**
20: **end if**
21: $\mathcal{O} = \mathcal{O} \cup$ *output*
22: **end while**
23: **end for**
24: **return** *TS-All*, *TS-Text*, *TS-Struct*, *TS-Seq*

statistical significance of the observed results. To answer **RQ2** the fault finding ability and sizes of the new test suites for the four definitions are compared. Test suite size is only an approximation of test effort. A larger test suite would require more time and effort to execute and maintain and to examine the output.

Experimental Set-up

The starting test suite used in the experiment is chosen as the test suite with the highest branch coverage from 30 test suites previously generated by SWAT as an input to a new tool SWAT-U that implements Algorithm 6. The test suite with the highest coverage is chosen to focus on additional fault finding ability of uniqueness, even where coverage is relatively high. The same SWAT test suite and mutation algorithm are used for random augmentation. Since the test data generation process is non-deterministic, the evaluation runs SWAT-U and random augmentation 30 times for each test suite, to support statistical significance testing. To detect faults, SWAT-U uses the automated oracle described in Section 1.5.2.

SWAT-U uses database tables to keep track of previous output. Execution times of SWAT-U are not reported in this experiment but the slowest run encountered was 20 minutes.

Results

Table 5.1 reports the results of running the approach 30 times on each application together with information about the original test suites. The new test cases increased coverage by only 1% or less for all applications with no additional coverage for FaqForge. The last column reports the improvement in percentage of each algorithm over random. This is calculated based on the faults per test case.

In three of the five applications, OU-All performed better than random augmentation. In Webchess no new faults were found by either the uniqueness approaches or random augmentation. This could be because no other faults exist or due to a limitation in the mutation algorithm. PHPSysInfo only has four user inputs which could limit the effect of user inputs on outputs.

As expected, OU-All adds the largest number of test cases to the test suite while OU-Seq adds the fewest. OU-Struct and OU-Text perform differently based on the application. In both PHPSysInfo and Timeclock, the number of unique outputs based on

Table 5.1: Results of average faults found and test suite size obtained from running the approach and random 30 times for each application. The (%) column in New Faults is calculated in relation to original faults found. Faults/Test results in bold perform statistically significantly better than random.

	Original				New	New Faults		Faults	Improv.
App Name	Cov	Tests	Faults	Algorithm	Tests	Num	%	/Test	on Rand
FAQForge	69%	36	50	Rand	180	3.6	7.1	0.020	-
				OU-All	180	5.0	9.9	**0.028**	40%
				OU-Text	96	1.3	2.7	0.014	-30%
				OU-Struct	170	5.0	9.9	**0.029**	48%
				OU-Seq	3	0.8	1.6	0.245	1137%
Schoolmate	70%	176	103	Rand	314	1.7	1.6	0.005	-
				OU-All	314	23.7	23.0	**0.076**	1324%
				OU-Text	120	21.4	20.8	**0.178**	3255%
				OU-Struct	252	20.3	19.7	**0.080**	1415%
				OU-Seq	32	9.9	9.6	**0.309**	5737%
Webchess	33%	49	70	Rand	299	0	0	0	-
				OU-All	299	0	0	0	0%
				OU-Text	54	0	0	0	0%
				OU-Struct	220	0	0	0	0%
				OU-Seq	2	0	0	0	0%
PHPSysInfo	22%	20	7	Rand	1138	2.1	30.0	0.002	-
				OU-All	1138	2.1	30.0	0.002	0%
				OU-Text	1138	2.1	30.0	0.002	0%
				OU-Struct	228	1.6	22.9	**0.007**	289%
				OU-Seq	8	1.7	24.8	**0.228**	12567%
Timeclock	21%	284	172	Rand	1366	2.7	1.6	0.002	-
				OU-All	1366	3.7	2.2	**0.003**	37%
				OU-Text	98	3.7	2.2	**0.038**	1812%
				OU-Struct	1317	2.7	1.6	0.002	2%
				OU-Seq	9	2.7	1.6	**0.307**	15408%

the strictest definition OU-All yield a comparatively large number of new test cases. Both of these applications contain a time-dependent element in the output: PHPSys-

Info prints the time the system has been available while Timeclock prints weather information. This reinforces the arguments for the need for different output uniqueness definitions.

OU-All also finds the largest number of faults with a few exceptions where another definition finds a matching number of faults with fewer test cases. In all applications, OU-Seq is the most efficient definition that found the largest number of faults per test case.

Answers to Research Questions

In this section the research questions posed at the start of Section 5.3.2 are answered, based on the empirical evidence from the experiments on the five web applications.

RQ1: Does using output-uniqueness augmented test suites enhance fault finding ability?

In four of the five applications studied, at least one output uniqueness definition performed better than random. In three applications, OU-All performed statistically significantly better than random. The best performing uniqueness definition significantly outperforms random augmentation in three of five applications by an overall average of 6,970% with one application (Webchess) where improvement was not possible.

RQ2: How do the four definitions of output uniqueness affect the fault finding ability and test effort of the generated test suite?

OU-All found the most faults overall but added the largest number of test cases. Less strict definitions result, in most cases, in a loss of some of the faults found by OU-All. OU-Seq performed the best in terms of faults per additional test case and shows the most potential.

5.4 Empirical Study: Output Uniqueness as a Test Adequacy Criteria

The experiment in Section 5.3 provided initial confirmation that output uniqueness criteria can be useful in increasing a test suite's effectiveness at finding faults. However, a more in-depth study is needed to investigate and understand the correlation between test suite effectiveness and the new output uniqueness criteria. This section provides an empirical study that analyses the correlation between various variables that affect a test suite's effectiveness: output uniqueness, structural coverage and size. This section also investigates the effectiveness of output uniqueness criteria as an alternative to structural coverage criteria when the source code cannot be instrumented either because it is unavailable or when the testing objective requires the code of the application to be unchanged as is required in stress testing.

5.4.1 Research Questions

This second empirical study in this chapter is designed to answer the following research questions:

RQ1: How do Output Uniqueness Criteria correlate with fault finding ability of a test suite?

This research question investigates how the uniqueness of output (according to the definitions proposed in Sections 5.2) correlates with the fault finding ability of test suites. It also examines how output uniqueness criteria compare to structural coverage criteria in terms of their comparative correlation with fault detection. To determine the usefulness of output uniqueness criteria, a strong correlation should exist between the number of distinct outputs in a test suite and the number of faults found by the test suite.

RQ2: How do Output Uniqueness Criteria correlate with structural coverage of a test suite?

This question investigates the correlation between output uniqueness criteria and structural coverage criteria. A strong correlation could indicate that output uniqueness cri-

teria can be used as alternative criteria when structural criteria are found to be inapplicable (e.g., when code is unavailable).

RQ3: How does the reliability of output uniqueness criteria compare to structural coverage criteria?

This research question compares the consistency in finding fault for both output uniqueness criteria and structural coverage criteria. The consistency of a criterion is denoted by the ability of every test suite that satisfies the criteria to find the same faults. This question also investigates whether output uniqueness criteria are complementary to structural coverage criteria or are a possible computationally cheaper alternative.

RQ4: How sensitive is the output to changes in input based on each output uniqueness definition?

This question investigates the sensitivity of the output to changes in the inputs for each output uniqueness definition. This question seeks to discover which patterns in changes in the output can provide an insight into how to integrate output uniqueness criteria in a test data generation process.

5.4.2 Experimental Design

This section describes the subjects, measures and analysis tools used in this empirical study.

Subjects

The web applications chosen for the empirical study are the six applications described in Section 1.5.1. For each application, all test cases generated from SWAT from Chapter 3 are collected to form a pool of test cases that are used for sampling for the experiments performed for this empirical study. Test cases are collected from those generated for each of the three algorithms implemented for SWAT. Only unique test cases are chosen: A unique test case is a test case that contains a request that has a unique URL and input-value pairs. Every test case is a sequence of one or two requests. That is, a test case consists of two requests when login is required for the action in the second request to be performed.

Table 5.2: Test data information: Number of test cases, faults found and total paths, branches and statements covered by the subject test data for each of the six web applications.

App Name	Test Cases	Faults	Coverage		
			Paths	Branches	Statements
FAQForge	7,233	67	176	96	516
Schoolmate	72,674	201	333	570	2,973
Webchess	9,377	98	209	460	2,097
PHPSysInfo	1,130	6	341	476	4,841
Timeclock	10,671	186	439	831	5,386
PHPBB2	22,379	79	2,030	1,337	7,807

Table 5.3: Output analysis information: the number of distinct outputs for each of the output uniqueness definitions.

App Name	Total Outputs	OU-						
		All	Text	Struct	Hidden	Subtypes	Seq	LastText
FAQForge	14,466	1,287	1,175	1,049	896	78	55	7
Schoolmate	145,348	1,982	638	1,489	1,464	326	271	248
Webchess	18,759	1,608	408	1,556	1,316	151	38	15
PHPSysInfo	1,130	1,047	1,047	707	14	14	14	884
Timeclock	21,342	3,014	782	2,638	249	111	80	89
PHPBB2	44,758	5,617	1,602	4,138	513	92	82	27

Measures

Coverage, fault detection and output uniqueness are measured and evaluated in the experiments conducted for the empirical study. For coverage: Path, branch and statement coverage are measured. Statement coverage and branch coverage are widely used in research for test data generation [SMA05, AKD+08, AKD+10, PLEB07, FK96, LMH10, WBP02]. Path coverage is a stronger criterion that subsumes both branch and statement coverage. Faults are detected and measured using the automated oracle described in Section 1.5.2.

The number of distinct outputs based on all output uniqueness definitions presented in Section 5.2 are also measured.

Table 5.2 provides information about the size of the test pool for each application, the total number of faults that can be detected, as well as the maximum paths, branches and statements covered by the test pool.

The pool of test data for Schoolmate is considerably larger than other applications (72k compared to 1-20k), while PHPSysInfo has the fewest test cases (1,130). However, these pools were not artificially reduced to a 'standard' size to avoid introducing bias into the experimental study.

Table 5.3 details the results of analyzing the output and measuring the numbers of distinct outputs for each of the output uniqueness criteria definitions. The table also shows the total number of outputs for the test pool which is equal to the number of requests in the pool rather than the number of test cases: a single test case may consist of multiple requests. Although all test cases in the pool are unique, the analysis shows that, even for the strictest output uniqueness definition, only a small percentage of outputs are distinct (ranging from 1-14%). The only exception is PHPSysInfo, for which nearly all outputs are unique for both the OU-All and OU-Text criteria. This is caused by the application displaying several data items that are time-sensitive on the output page: Execution time and system up time.

The analysis also shows that web applications behave differently in the number of distinct unique outputs for each definition and how these numbers relate. For example, for PHPSysinfo the number of distinct outputs for the OU-All and OU-text criteria is the same while for other applications the number of outputs for OU-Text is considerably lower than OU-All. On the other hand, the number of outputs for OU-Hidden is similar to the number for OU-Struct for FaqForge, Schoolmate and Webchess while for PHPSysInfo and Timeclock OU-Hidden reduces the number of distinct outputs considerably. These results give insight into where the variation in output lies for the different web applications used in this study.

Analysis Tools

The web applications' code was instrumented to record branch and path coverage. Xdebug was used to record statement coverage. Xdebug cannot be used for branch and path coverage because it produces a list of filenames and line numbers that were executed

rather than a trace. The list does not reflect order or frequency of execution. A test harness was developed to execute test cases and analyse the output. Statistical analysis and data visualization were performed using the statistical package R.[1]

5.4.3 Experiments and Discussion

This section describes the experimental set up and discusses the results obtained for each of the analyses performed to answer each of the research questions posed.

Correlation with Fault Finding Ability

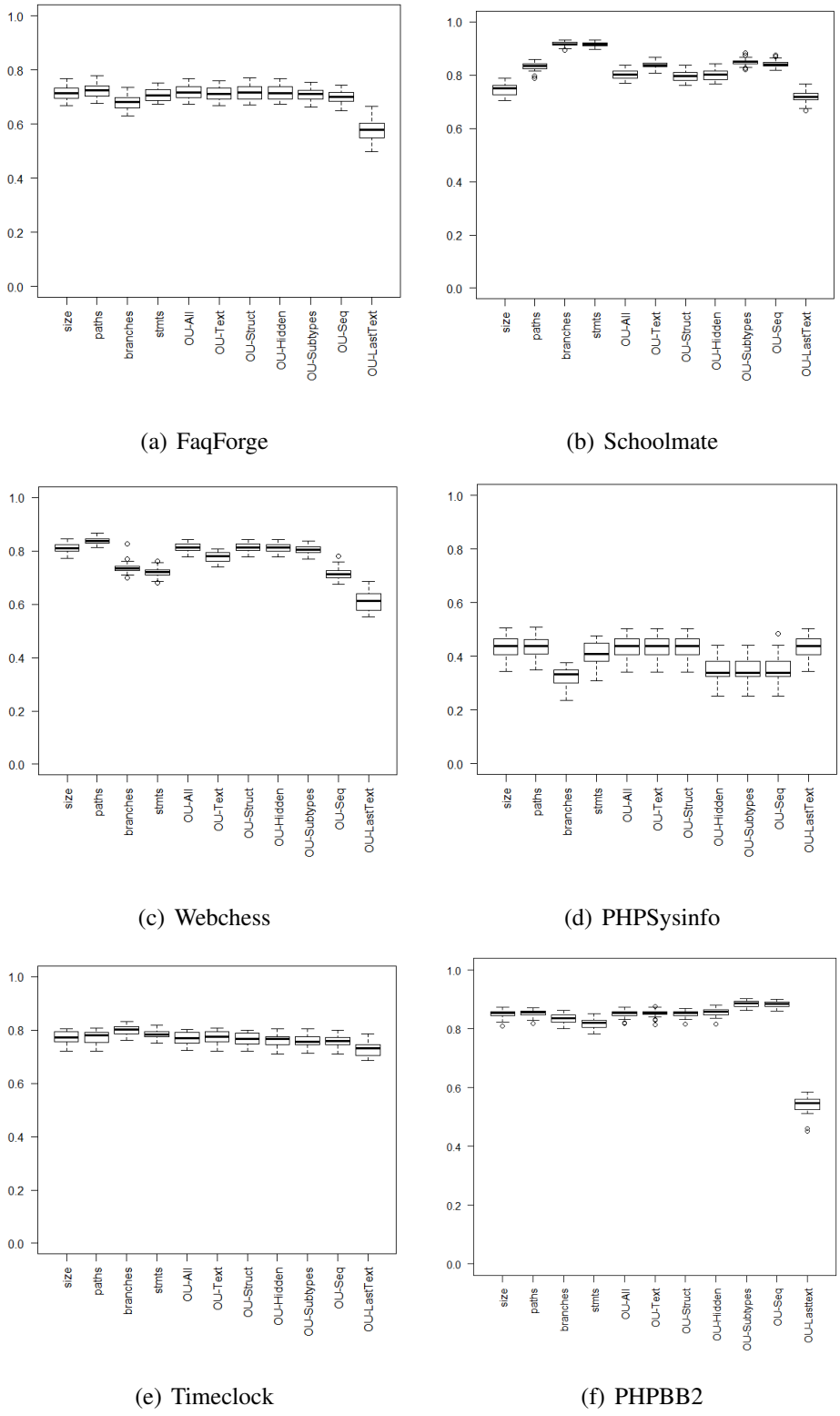
To investigate the correlation between fault finding ability of a test suite and structural coverage and output uniqueness criteria, a set of 500 test suites was randomly sampled from the test pool for each application. The number of test cases in each test suite was randomly selected between 10 and 500 test cases. These upper and lower limits were selected to be an order of magnitude smaller/larger than the smallest/largest test suite in the original test suites generated by SWAT. For each test suite, the values of all output uniqueness criteria, structural coverage, faults found and size were recorded. Spearman's rank correlation coefficient was then calculated for the set. Since the sampling process is random, the experiment was repeated 30 times for each application.

A set size of 500 test suites was chosen after experimenting with three test suite set sizes: 20, 100 and 500. These three test suite set sizes were examined to investigate the effect of the number of test suites sampled on results and to gain more confidence in the discovered correlations between fault finding ability and coverage and output uniqueness criteria. In total 18,600 randomly selected test suites were sampled for each application.

Spearman's rank correlation coefficient was chosen because it is non-parametric and since the sampling process of test suites is random, no assumptions about the distribution of faults found, coverage or output uniqueness can be made.

Figure 5.7 reports the results of the analysis for the six web applications studied. For all applications but PHPSysInfo and all experiments, the results indicate that a strong correlation exists between output uniqueness criteria and fault finding ability that is comparable to the correlation between coverage and fault finding ability. The

[1] http://www.r-project.org/



(a) FaqForge (b) Schoolmate

(c) Webchess (d) PHPSysinfo

(e) Timeclock (f) PHPBB2

Figure 5.7: Variations in Spearman's rank correlation coefficient for test suites sets of size 500 over 30 different experiments for the six applications.

results also indicate that OU-LastText is less well correlated with fault finding ability than all other output and coverage criteria for all applications.

For PHPSysInfo, a correlation exists between fault finding ability and output uniqueness. However, the correlation is not as strong as the correlation observed for other applications. This weaker correlation can also be observed between structural coverage criteria and fault detection. The quality of the test pool for PHPSysInfo might have an influence on results because the number of maximum faults that can be found by all test cases is considerably smaller than all other applications.

While all criteria are correlated with fault finding ability for FaqForge, TimeClock and PHPBB2 in a similar manner, the correlations vary more for Schoolmate and Webchess. For Schoolmate, branch and statement coverage have the highest correlation with fault detection. For Webchess, path coverage has the strongest correlation with fault finding ability while branch and statement coverage are less correlated with fault finding ability than most output uniqueness criteria.

The coverage criteria measured are, by definition, linearly ordered: Path coverage subsumes branch coverage and branch coverage subsumes statement coverage [Zhu95]. For some applications in the experiments, a weaker coverage criteria has a stronger correlation with fault finding ability than some other criterion that subsumes it. This can be explained by the possibility that a test suite can cover more paths or branches without finding new faults, thereby affecting the correlation of that criterion with fault finding ability.

The results of using a test suite set size of 20 and 100 can be viewed in Figures A.1 and A.2 in Appendix A. The results show that increasing the number of test suites in a set reduces the variation in Spearman's rank correlation coefficient; as expected more data give tighter bounds on observed correlations. However, the medians were similar for all three sizes and the relationships between different criteria were maintained. Based on these results, increasing the number of test suites per set might further reduce the variation in Spearman's rank correlation coefficient but there is sufficient evidence that the conclusions that can be drawn from the 500 test suite sets are likely to remain robust.

Correlation with Structural Coverage

To investigate the correlation between structural coverage criteria and output uniqueness criteria, Spearman's rank correlation coefficient was calculated between each of the three structural coverage measures and the number of distinct outputs for each of the output uniqueness definitions for each set of test suites.

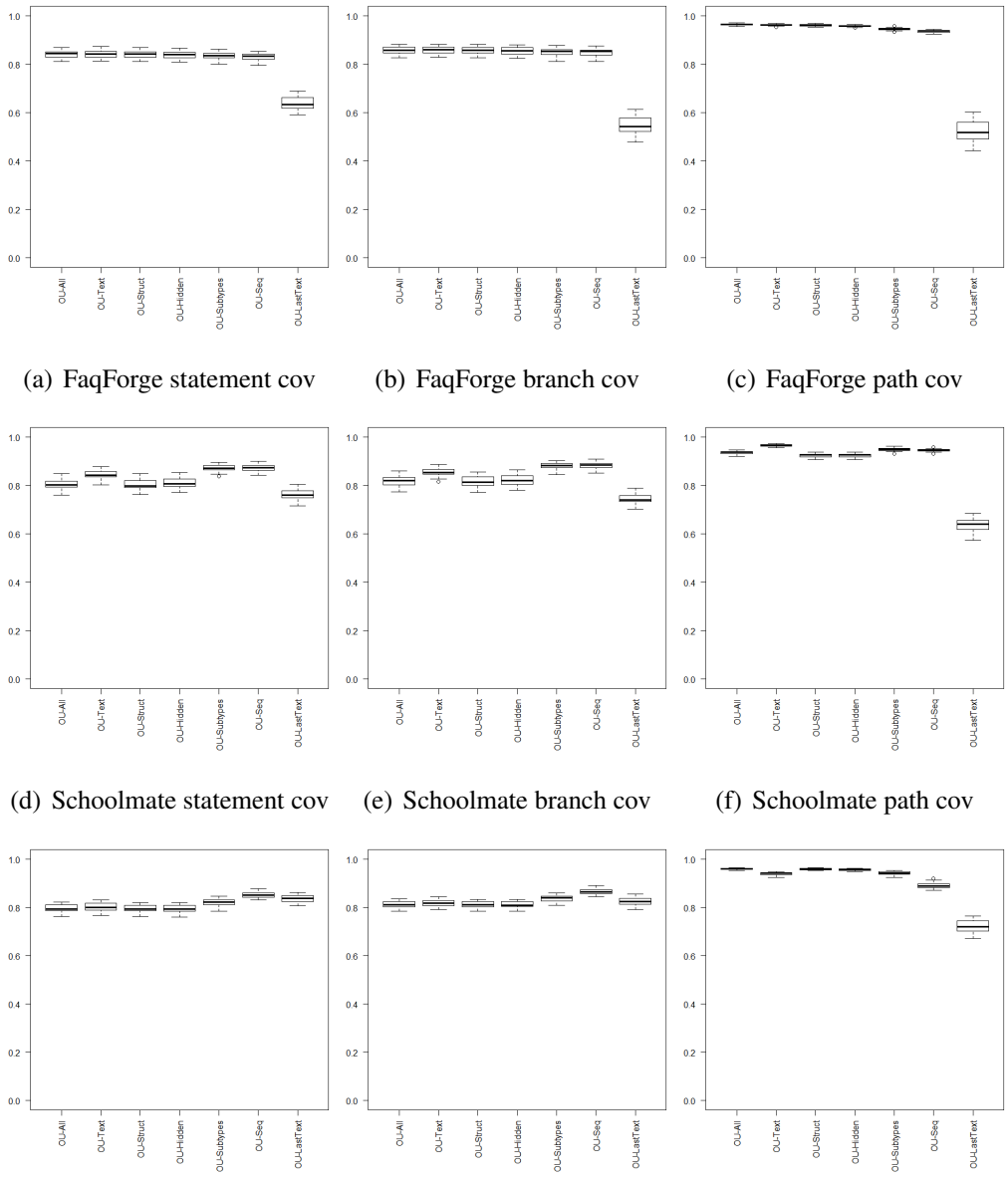
Figures 5.8 and 5.9 present the results of the analysis. The box plots in each row show the variations in Spearman's rank correlation coefficient for the 30 sets of 500 test suites for each application. The column of first box plots shows the results for statement coverage while the second shows branch coverage and the third shows path coverage. The box plots for test suite set sizes of 20 and 100 are in Appendix A.

Over all applications, a strong correlation can be observed between structural coverage and output uniqueness except for OU-LastText. OU-LastText has the weakest correlation with structural coverage for all applications except for PHPSysInfo. For PHPSysInfo, OU-LastText is strongly correlated with coverage which can be caused by the relatively large number of distinct OU-LastText outputs observed compared to other applications. Also, for PHPSysInfo, OU-Hidden, OU-Subtypes and OU-Seq have the weakest correlation with structural coverage.

The results also show that output uniqueness has a stronger correlation with path coverage than with statement and branch coverage for all applications. Statement and branch coverage have similar correlations with output uniqueness for all applications except PHPSysInfo where statement coverage has a stronger correlation with output uniqueness than branch coverage.

Reliability and Complementarity To Structural Coverage

To investigate the reliability of the proposed output uniqueness criteria, the consistency of finding faults for each distinct output (based on each output uniqueness definition) was analysed with regards to the test pool. If all test cases that produce the same output always find the same faults, these faults are guaranteed to be detected by any test suite constructed from the test pool that satisfies the output uniqueness criteria. The same analysis is also conducted for structural coverage to compare the reliability of output uniqueness criteria to structural coverage which is considered the state-of-the-art.



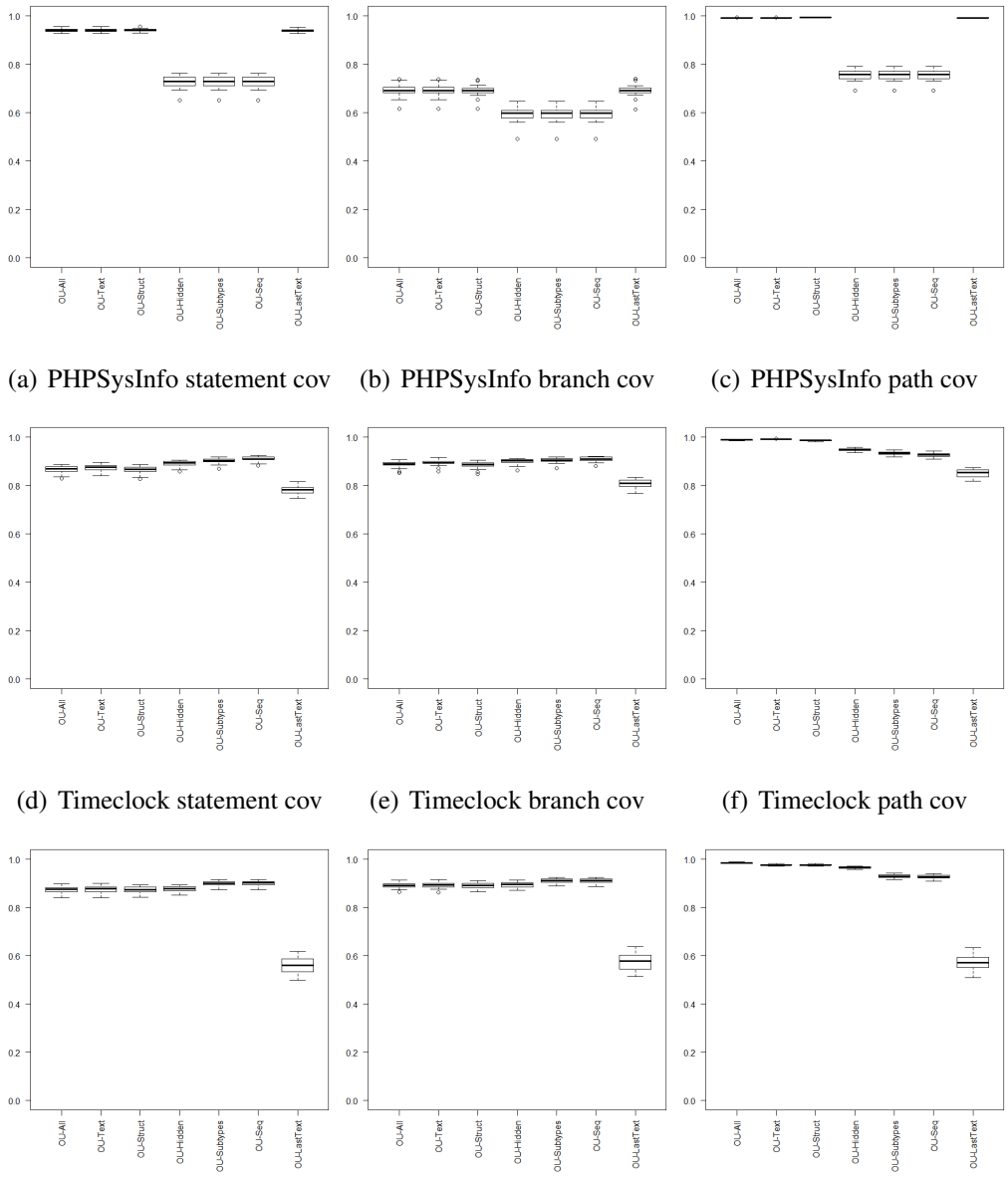
(a) FaqForge statement cov (b) FaqForge branch cov (c) FaqForge path cov

(d) Schoolmate statement cov (e) Schoolmate branch cov (f) Schoolmate path cov

(g) Webchess statement cov (h) Webchess branch cov (i) Webchess path cov

Figure 5.8: Variations in Spearman's rank correlation coefficient between structural coverage and output uniqueness for test suites sets of size 500 over 30 different experiments for FaqForge, Schoolmate and Webchess.

The homogeneity of faults consistently found by output uniqueness criteria is also compared to structural coverage criteria to investigate whether the two criteria types are complementary. As mentioned before, statement and branch coverage are widely used as test adequacy criteria. Path coverage was chosen for the comparison because it is the strongest of the three structural coverage criteria (according to the test adequacy subsumption relation) considered and therefore should output uniqueness criteria be found



(a) PHPSysInfo statement cov (b) PHPSysInfo branch cov (c) PHPSysInfo path cov

(d) Timeclock statement cov (e) Timeclock branch cov (f) Timeclock path cov

(g) PHPBB2 statement cov (h) PHPBB2 branch cov (i) PHPBB2 path cov

Figure 5.9: Variations in Spearman's rank correlation coefficient between structural coverage and output uniqueness for test suites sets of size 500 over 30 different experiments for PHPSysInfo, Timeclock and PHPBB2.

to be complementary to path coverage, they can be expected to be complementary to other structural coverage criteria not considered in this study.

Table 5.4 reports the results of the analysis for statement coverage. The number reported for consistent statement faults (Con) denotes the number of faults that are detected by all test cases in the test pool that cover a certain statement. That is, if a test suite is randomly constructed from the test pool that covers all possible statements, the faults are guaranteed to be revealed. In a similar way, consistent output uniqueness

faults are the number of faults that are detected by all test cases that produce a certain output. That is, if a test suite is randomly constructed that produces all distinct outputs in the pool, the faults are guaranteed to be revealed.

The results show that for all applications but PHPBB2 at least OU-All, OU-Text and OU-Struct find more faults consistently than statement coverage. In PHPBB2, only OU-All finds faults more consistently than statement coverage. Examining the faults for each approach shows that output uniqueness criteria and statement coverage are complementary as can be seen in the number of faults that are consistently found only by statement coverage ($S-O$) and only by output uniqueness criteria ($O-S$). Similar results can be observed for branch coverage in Table 5.5. These results suggest that output uniqueness criteria are more reliable than both branch and statement coverage and that the two types of criteria are complementary.

Table 5.6 shows the results of the analysis for path coverage. For all applications but Webchess and PHPSysInfo, path coverage is more consistent than all output uniqueness criteria. However, when comparing the consistent faults for output to those for path coverage, in four of the six applications output uniqueness is complementary to path coverage. For Schoolmate, Webchess and PHPSysInfo, even the weakest output uniqueness definition (OU-LastText) adds value to path coverage in fault finding.

For all applications but PHPSysInfo, a subset of faults is found consistently by neither path coverage nor output uniqueness. This suggests that a hybrid combining structural coverage and output uniqueness into a single criterion might be able to reveal those faults consistently. For example, a new criterion could be defined to select test cases that cover all paths in a test pool and for each path select one test case for each unique output.

The faults detected consistently for output uniqueness criteria for OU-All are comparable to path coverage across all applications. The number of faults found consistently is reduced compared to path coverage by a maximum of 14% over all applications. OU-Text also consistently finds faults across all applications for which results are competitive with OU-All. Since output uniqueness criteria does not require the availability of source code, this finding suggests that output uniqueness criteria can be effective as test selection/adequacy criteria when information about structural coverage cannot be obtained.

Table 5.4: Consistent fault finding ability for each output uniqueness criteria and homogeneity analysis compared to statement coverage consistently found faults.

App Name	All Faults (F)	Statement Faults (S) Con	Statement Faults (S) Incon	OU-	Output Faults (O) Con	Output Faults (O) Incon	$S \cup O$	$S \cap O$	$S - O$	$O - S$	$F - (S \cup O)$
FAQForge	67	38	29	All	55	12	58	35	3	20	9
				Text	51	16	57	32	6	19	10
				Struct	55	12	58	35	3	20	9
				Hidden	52	15	55	35	3	17	12
				Subtypes	48	19	54	32	6	16	13
				Seq	48	19	54	32	6	16	13
				LastText	13	54	38	13	25	0	29
Schoolmate	201	78	123	All	162	39	168	72	6	90	33
				Text	134	67	145	67	11	67	56
				Struct	148	53	157	69	9	79	44
				Hidden	139	62	151	66	12	73	50
				Subtypes	105	96	132	51	27	54	69
				Seq	84	117	122	40	38	44	79
				LastText	14	187	85	7	71	7	116
Webchess	98	30	68	All	77	21	77	30	0	47	21
				Text	56	42	58	28	2	28	40
				Struct	71	27	77	24	6	47	21
				Hidden	64	34	75	19	11	45	23
				Subtypes	59	39	70	19	11	40	28
				Seq	17	81	32	15	15	2	66
				LastText	19	79	31	18	12	1	67
PHPSysInfo	6	4	2	All	6	0	6	4	0	2	0
				Text	6	0	6	4	0	2	0
				Struct	5	1	6	3	1	2	0
				Hidden	3	3	4	3	1	0	2
				Subtypes	3	3	4	3	1	0	2
				Seq	3	3	4	3	1	0	2
				LastText	2	4	6	0	4	2	0
Timeclock	186	79	107	All	139	47	147	71	8	68	39
				Text	125	61	133	71	8	54	53
				Struct	90	96	106	63	16	27	80
				Hidden	55	131	82	52	27	3	104
				Subtypes	51	135	81	49	30	2	105
				Seq	50	136	81	48	31	2	105
				LastText	27	159	79	27	52	0	107
PHPBB2	79	69	10	All	70	9	75	64	5	6	4
				Text	55	24	75	49	20	6	4
				Struct	61	18	75	55	14	6	4
				Hidden	48	31	75	42	27	6	4
				Subtypes	45	34	75	39	30	6	4
				Seq	43	36	75	37	32	6	4
				LastText	12	67	72	9	60	3	7

Table 5.5: Consistent fault finding ability for each output uniqueness criteria and homogeneity analysis compared to branch coverage consistently found faults.

App Name	All Faults (F)	Branch Faults (B) Con	Branch Faults (B) Incon	OU-	Output Faults (O) Con	Output Faults (O) Incon	$B \cup O$	$B \cap O$	$B - O$	$O - B$	$F - (B \cup O)$
FAQForge	67	37	30	All	55	12	58	34	3	21	9
				Text	51	16	57	31	6	20	10
				Struct	55	12	58	34	3	21	9
				Hidden	52	15	55	34	3	18	12
				Subtypes	48	19	54	31	6	17	13
				Seq	48	19	54	31	6	17	13
				LastText	13	54	37	13	24	0	30
Schoolmate	201	102	99	All	162	39	176	88	14	74	25
				Text	134	67	154	82	20	52	47
				Struct	148	53	168	82	20	66	33
				Hidden	139	62	163	78	24	61	38
				Subtypes	105	96	145	62	40	43	56
				Seq	84	117	135	51	51	33	66
				LastText	14	187	103	13	89	1	98
Webchess	98	44	54	All	77	21	77	44	0	33	21
				Text	56	42	58	42	2	14	40
				Struct	71	27	77	38	6	33	21
				Hidden	64	34	75	33	11	31	23
				Subtypes	59	39	70	33	11	26	28
				Seq	17	81	46	15	29	2	52
				LastText	19	79	45	18	26	1	53
PHPSysInfo	6	4	2	All	6	0	6	4	0	2	0
				Text	6	0	6	4	0	2	0
				Struct	5	1	6	3	1	2	0
				Hidden	3	3	4	3	1	0	2
				Subtypes	3	3	4	3	1	0	2
				Seq	3	3	4	3	1	0	2
				LastText	2	4	6	0	4	2	0
Timeclock	186	136	50	All	139	47	163	112	24	27	23
				Text	125	61	149	112	24	13	37
				Struct	90	96	163	63	73	27	23
				Hidden	55	131	139	52	84	3	47
				Subtypes	51	135	138	49	87	2	48
				Seq	50	136	138	48	88	2	48
				LastText	27	159	136	27	109	0	50
PHPBB2	79	74	9	All	70	9	74	66	4	4	5
				Text	55	24	74	51	19	4	5
				Struct	61	18	74	57	13	4	5
				Hidden	48	31	74	44	26	4	5
				Subtypes	45	34	74	41	29	4	5
				Seq	43	36	74	39	31	4	5
				LastText	12	67	71	11	59	1	8

Table 5.6: Consistent fault finding ability for each output uniqueness criteria and homogeneity analysis compared to path coverage consistently found faults.

App Name	All Faults (F)	Path Faults (P) Con	Path Faults (P) Incon	OU-	Output Faults (O) Con	Output Faults (O) Incon	$P \cup O$	$P \cap O$	$P - O$	$O - P$	$F - (P \cup O)$
FAQForge	67	62	5	All	55	12	62	55	7	0	5
				Text	51	16	62	51	11	0	5
				Struct	55	12	62	55	7	0	5
				Hidden	52	15	62	52	10	0	5
				Subtypes	48	19	62	48	14	0	5
				Seq	48	19	62	48	14	0	5
				LastText	13	54	62	13	49	0	5
Schoolmate	201	186	15	All	162	39	192	156	30	6	9
				Text	134	67	190	130	56	4	11
				Struct	148	53	191	143	43	5	10
				Hidden	139	62	191	134	52	5	10
				Subtypes	105	96	189	102	84	3	12
				Seq	84	117	187	83	103	1	14
				LastText	14	187	187	13	173	1	14
Webchess	98	76	22	All	77	21	83	69	7	7	15
				Text	56	42	79	53	23	3	19
				Struct	71	27	82	65	11	6	16
				Hidden	64	34	80	60	16	4	18
				Subtypes	59	39	78	57	19	2	20
				Seq	17	81	76	17	59	0	22
				LastText	19	79	77	18	58	1	21
PHPSysInfo	6	4	2	All	6	0	6	4	0	2	0
				Text	6	0	6	4	0	2	0
				Struct	5	1	6	3	1	2	0
				Hidden	3	3	4	3	1	0	2
				Subtypes	3	3	4	3	1	0	2
				Seq	3	3	4	3	1	0	2
				LastText	2	4	6	0	4	2	0
Timeclock	186	162	24	All	139	47	166	135	27	4	20
				Text	125	61	165	122	40	3	21
				Struct	90	96	166	86	76	4	20
				Hidden	55	131	162	55	107	0	24
				Subtypes	51	135	162	51	111	0	24
				Seq	50	136	162	50	112	0	24
				LastText	27	159	162	27	135	0	24
PHPBB2	79	76	3	All	70	9	76	70	6	0	3
				Text	55	24	76	55	21	0	3
				Struct	61	18	76	61	15	0	3
				Hidden	48	31	76	48	28	0	3
				Subtypes	45	34	76	45	31	0	3
				Seq	43	36	76	43	33	0	3
				LastText	12	67	76	12	64	0	3

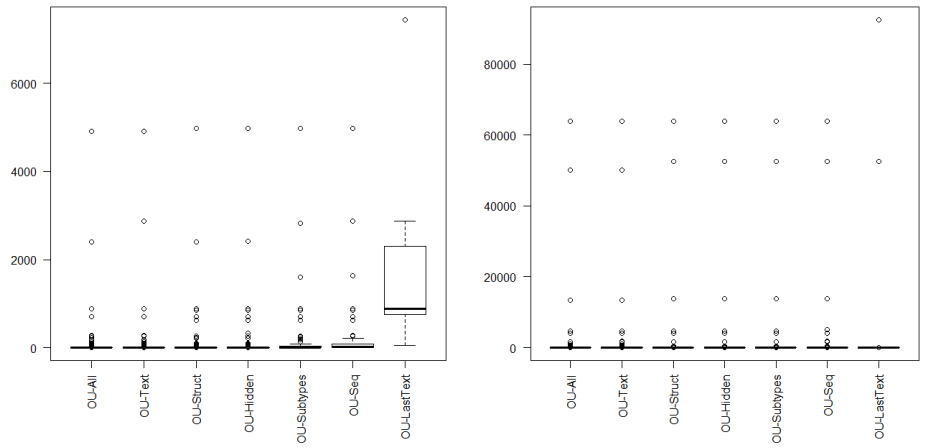
Output Sensitivity to Input

The pool of test data for each application was analysed to investigate patterns in the numbers of distinct outputs observed for each output definition and the frequency for which each distinct output appears.

Figure 5.10 shows the variation in the frequency for which each distinct output is observed for each output uniqueness definition and each application. The results show a similar pattern over all applications: Each distinct output is observed only a small number of times for each application while a few outputs are observed a very high number of times (represented by the outliers in the box plots).

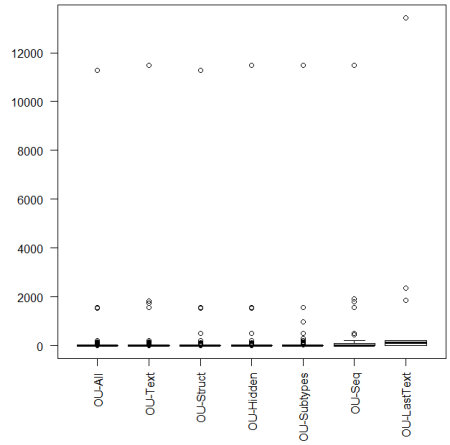
Table 5.7 reports further details from the analysis: The column **Distinct** reports the number of distinct outputs observed for each definition and also the percentage of those distinct outputs in relation to the total number of outputs. The results show that for five of the six applications, the number of distinct outputs is relatively low compared to the total number of outputs (maximum of 14%). The percentage is lowest for Schoolmate (1.4%) which has the largest test pool (72k). This suggests that it could be feasible to use output uniqueness as a test adequacy criterion because the number of unique outputs might grow manageably. However, this observation does not hold for PHPSysInfo where almost 93% of the observed outputs are distinct for OU-All and OU-Text. As mentioned previously, this is caused by the application printing time-sensitive data at the bottom of the output page which can be also observed in the high number of distinct OU-LastText.

The table also reports the number of outputs that were only observed in one test case. For all applications and most output uniqueness definitions, a high percentage of outputs are only observed once. This is especially noticeable for OU-All, OU-Text and OU-Struct, but can also be observed for other output uniqueness definitions. Surprisingly, even for Schoolmate where the test pool is relatively large compared to other applications, a large percentage of distinct outputs was only observed once. These changes in output that can only be observed once might not reflect interesting application behaviour. For example, consider a test case that edits a record where the output page displays the record after the changes. Any other test case that edits the same record but with different input values will have a different output. However, the application behaviour being tested by all such cases is essentially the same.

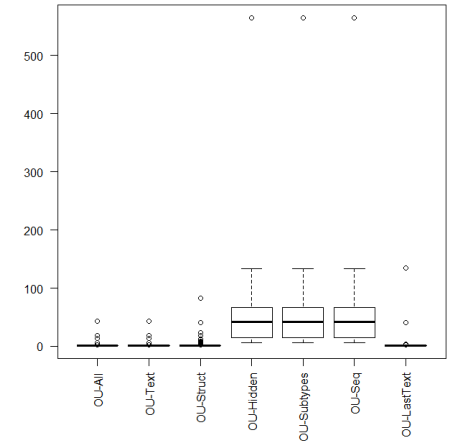


(a) FaqForge

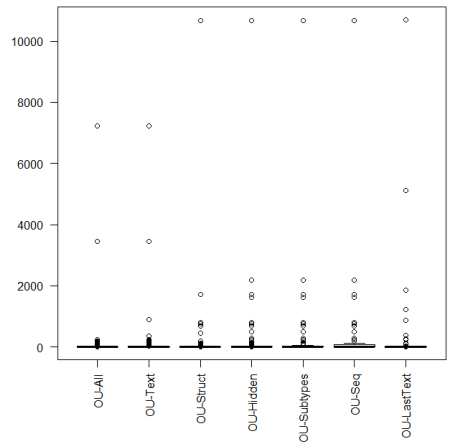
(b) Schoolmate



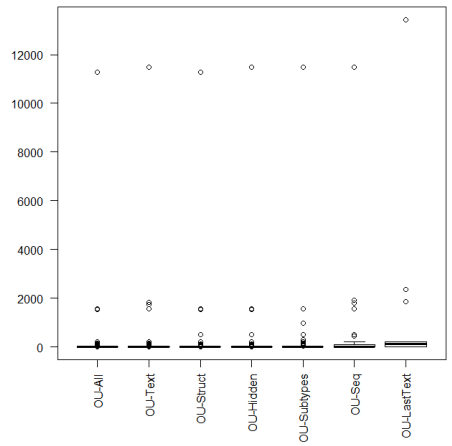
(c) Webchess



(d) PHPSysinfo



(e) Timeclock



(f) PHPBB2

Figure 5.10: Variation in frequency of observing each distinct output for each output uniqueness definition for all six applications studied.

These results suggest that two main patterns can be observed in the manner in which the output is affected by changes in inputs: Some changes in inputs have no effect on the output while other changes have a very strong effect on output. These observations can be useful in designing a test generation method that is based on output uniqueness. The test data generator can dynamically discover inputs that have no effect on output or have a high effect on output and prioritize mutating inputs that do not fall into one of those two categories. The generator might also construct an application-specific output uniqueness definition dynamically by clustering the observed outputs.

5.4.4 Answers to Research Questions

This section presents summarized answers to the research questions based on the results of the experiments and statistical analysis of the results.

RQ1: How do Output Uniqueness Criteria correlate with fault finding ability of a test suite?

The results showed, with high confidence levels (p-value $<$ 2.20e-16), that for five of the six applications and for all output uniqueness criteria except OU-LastText, a strong correlation exists between output uniqueness and fault finding ability using Spearman's rank correlation. For all six applications, this correlation is as strong as the correlation between fault finding and structural coverage criteria.

RQ2: How do Output Uniqueness Criteria correlate with structural coverage of a test suite?

For all six applications, the results showed, with high confidence levels (p-value $<$ 2.20e-16), that a strong correlation exists between all output uniqueness criteria except OU-LastText and structural coverage criteria. Path coverage has the strongest correlation with output uniqueness criteria. These results suggest that output uniqueness criteria can be effective as an alternative to structural coverage criteria, when they are inapplicable.

RQ3: How does the reliability of output uniqueness criteria compare to structural coverage criteria?

For all applications, test suites that satisfy OU-All, OU-Text and OU-Struct criteria find more faults consistently than both statement and branch coverage. All of the out-

Table 5.7: Output sensitivity to changes in input values.

App Name	All Outputs	OU-	Distinct Num	Distinct %	Observed once Num	Observed once %
FAQForge	14,466	All	1,287	8.9	942	73.2
		Text	1,175	8.1	877	74.6
		Struct	1,049	7.3	718	68.4
		Hidden	896	6.2	626	69.9
		Subtypes	78	1.0	13	16.7
		Seq	55	0.4	8	14.5
		LastText	7	0.1	0	0.0
Schoolmate	145,348	All	1,982	1.4	1,359	68.6
		Text	638	0.4	486	76.2
		Struct	1,489	1.0	899	60.4
		Hidden	1,464	1.0	888	60.7
		Subtypes	326	0.2	214	65.6
		Seq	271	0.2	174	64.2
		LastText	248	0.2	241	97.2
Webchess	18,759	All	1,608	8.6	1,226	76.2
		Text	408	2.2	279	68.4
		Struct	1,556	8.3	1,158	74.4
		Hidden	1,316	7.0	948	72.0
		Subtypes	151	0.8	58	38.4
		Seq	38	0.2	5	13.2
		LastText	15	0.1	2	13.3
PHPSysInfo	1,130	All	1,047	92.7	1041	99.4
		Text	1,047	92.7	1041	99.4
		Struct	707	62.6	582	82.3
		Hidden	14	1.2	0	0
		Subtypes	14	1.2	0	0
		Seq	14	1.2	0	0
		LastText	15	1.3	816	92.3
Timeclock	21,342	All	3,014	14.1	2,235	74.2
		Text	782	3.7	420	53.7
		Struct	2,638	12.4	1,963	74.4
		Hidden	249	1.2	159	63.9
		Subtypes	111	0.5	24	21.6
		Seq	80	0.4	6	7.5
		LastText	89	0.4	61	68.5
PHPBB2	44,758	All	5,617	12.6	4,626	82.4
		Text	1,602	3.6	646	40.3
		Struct	4,138	9.3	4,019	97.1
		Hidden	513	1.2	287	55.9
		Subtypes	92	0.2	30	32.6
		Seq	82	0.2	25	30.5
		LastText	27	0.1	19	70.4

put uniqueness criteria are also found to be complementary to branch and statement coverage.

For Webchess and PHPSysInfo, OU-All consistently finds more faults than path coverage. For all other applications, OU-All reduces the fault detection ability by a maximum of 14%. The number of faults found consistently by output uniqueness criteria decreases as the criteria become less strict with OU-TextLast being the least effective in consistently finding faults across all applications. These results confirm that the more strict criteria can be effective as test adequacy criteria when structural coverage criteria cannot be measured.

For four of the six applications, the faults found consistently by output uniqueness criteria are complementary to those found by path coverage.

RQ4: How sensitive is the output to changes in input based on each output uniqueness definition?

The results of the analysis revealed an interesting pattern that can be observed over all applications. Although each test case has a unique set of inputs, only a small percentage of outputs are distinct, even for the strictest output uniqueness definition (OU-All). On the other hand, a majority of distinct outputs can only by observed once for OU-All, OU-Struct, OU-Text and OU-Hidden. This suggests that the output is not sensitive to changes in some inputs, while it is highly sensitive to changes in other inputs. This observation can be used to dynamically guide a test generation process that is based on output uniqueness in selecting which inputs to mutate to find interesting distinct outputs. It can be expected that inputs that lie in between those two extremes denote interesting behaviour in the application's execution.

5.4.5 Threats to Validity

Internal threats: Internal threats to validity are factors that affect the dependent variables and are not controlled in the experiments. The test suites generated for RQ1 and RQ2 were all generated randomly in the same manner. The choice of test suite size might have an effect on results. To address this, a random size between 10 and 500 was chosen because test suites smaller than 10 might not display a diversity in coverage and number of distinct output and 500 is larger than the largest test suite size that was originally produced by a single run of SWAT.

External threats: External threats to validity are threats that limit the ability to generalize results. There are three main threats to external validity: the choice of applications studied, the fault oracle used and the source of test cases in the test pools. The number of applications studied is naturally limited. An empirical study of more applications is needed before being able to generalize results. However, the selected applications represent different domains and are used by real users. They also have diverse sizes and architectures.

The faults measured are faults that can be detected automatically. To generalize results to other types of faults, an investigation of how the automated faults relate to other types of faults is needed. However, the faults reported and used in this study are, nevertheless, real faults.

The test cases in the test pool were all generated for branch coverage using SWAT. This can affect the results and the ability to generalize them to test cases generated to achieve other objectives and using other tools. However, branch coverage is the standard widely used to generate test suites. The proposed new criteria are proposed to be used in conjunction with branch coverage or as an alternative. Also, using test cases that were generated for branch coverage might introduce a bias related to branch coverage.

Construct threats: Construct threats are related to the measures used in the experiments and their ability to capture what they are measuring. Fault finding ability was selected to measure the effectiveness of test suites and the different criteria because it is the aim of any functional testing process. Path, branch and statement coverage were selected to represent structural coverage because branch and statement coverage are widely used in the literature [SMA05, AKD+08, AKD+10, PLEB07, FK96, LMH10, WBP02] while path coverage is the strongest structural coverage criteria and thereby subsumes other structural criteria.

5.5 Related Work

Raghavan and Garcia-Molina [RGM01] developed a crawler (HiWE) that uses the output to enhance the crawling process. The output is analysed to identify pages that appear frequently in response to a `form` submission and mark them as error pages. The crawler's goal is to extract data from a web application through its interface and

therefore the approach proposed is used to identify whether a request returned a new 'record'. The approach proposed in this chapter aims to use output as a test adequacy criterion and therefore is concerned with interesting executions of the application that can be identified through the output rather than attempting to retrieve more records. More details about HiWE were provided on page 49 of the literature review.

Sampath et al. [SBV+08] used HTML output to automate the oracle for regression testing (page 68 of the literature review) while Di Lucca [DLDPF02] used it to detect clones in static web pages. Some of the output definitions proposed in Sections 5.2 use some concepts of categorizing HTML output and resolving issues in dynamic content that these previous research papers used. However, those concepts are applied to test case generation rather than oracle automation and clone detection.

Artzi et al. [ADTP10] used a path constraint similarity criterion to generate test cases to localize faults with minimal test cases. Their approach generates additional test cases for previously known faults for localization and debugging, while the criteria proposed in this chapter aims to generate test cases to find new faults.

In feedback directed random testing [PLEB07, PLB08], the result of executing a test case is observed to exclude test cases that cause violations and exceptions from future generations. GUI feedback directed testing [YM10] also observes the effect of a test case on the state to generate new sequences using interacting events. These approaches, discussed in more detail in Section 2.6, use the effects on state to generate test sequences while the output criteria proposed in this chapter are used to select test cases that could be effective.

5.6 Conclusion

This chapter proposed a new test generation criterion based on output uniqueness. In the preliminary evaluation, this new criterion proved to be useful in finding test cases that can reveal new faults. A more in-depth empirical study provided evidence and support to the claim that the number of distinct outputs observed in a test suite is strongly correlated with both fault finding ability and structural coverage. An analysis of the reliability of the proposed output uniqueness criteria concluded that output uniqueness criteria performs better than both statement and branch coverage and is complementary to all structural coverage criteria including path coverage.

These results suggest that the new output uniqueness criteria can be effective as complementary criteria to structural coverage. The results also suggest that output uniqueness can also be used as an alternative criterion when all or part of the application code is unavailable for instrumentation as, for example, in cases where an application uses third party components.

The empirical study also provided insight into the effect of input changes on the output which could guide the design of a test generation method that is based on output uniqueness.

Chapter 6

Conclusion and Future Work

This chapter summarizes the overall conclusions of this thesis and how the work presented addressed the objectives it aimed to investigate. It also discusses how the approaches presented can be extended and enhanced in future work.

6.1 Conclusion

The aim of this thesis was to test the server-side code of a web application and to investigate the use of output in enhancing server-side testing.

In summary, the objectives of this thesis, as stated in the introduction, are the following:

1. Applying search based test data generation algorithms to web applications while extending them to handle web-specific challenges and to utilize output in the search process.

2. Investigating the use of dataflow analysis of the application server-side state in test suite regeneration and the use of server-side output to enhance the test sequence generation process.

3. Investigating the effectiveness of using output uniqueness as a test selection criterion for test suite augmentation.

4. Empirically investigate the correlation between output uniqueness and test suite effectiveness and between output uniqueness and structural coverage.

The first objective was addressed in Chapter 3, which adapted and implemented a search based algorithm for test data generation for PHP web applications in a tool

SWAT. SWAT addressed Web and PHP specific challenges, such as dynamic typing, identifying the interface and automatically discovering top level files without relaying only on dynamic analysis that might be incomplete. Chapter 3 also introduced a novel algorithm that uses the output of executed test cases as well as dynamic data collected during execution by seeding dynamically mined values from those sources into the search space. The empirical results confirmed that using this dynamic seeding approach (**DMV**) improved both effectiveness and efficiency of the generated test suites.

Chapter 4 addressed the second objective by using dataflow analysis of session variables and database tables to generate test sequences that are effective in increasing coverage and revealing new faults. The value-aware approach VADU enhances dataflow analysis by using values, in session variables and database query strings, to guide the generation of test sequences. Chapter 4 also introduced a tool SART that implements the proposed approaches. The evaluation showed that this value-aware approach is more effective in finding faults and increasing coverage than random regeneration. The approach also overcomes the limitation of static analysis in identifying Def-Use pairs for database tables when SQL statements are constructed dynamically.

The experiment in Section 5.3 addressed the third objective by presenting an approach that augmented test suites with test cases that produce new outputs not observed in the output of the original test suite. A tool SWAT-U was presented that implements the approach. The evaluation indicated that this approach produced new augmented test suite that are more effective in finding faults.

The last objective was addressed in Section 5.4: The results of the empirical analysis showed a strong correlation between output uniqueness and fault detection as well as between output uniqueness and structural coverage. The results also showed that output uniqueness is complementary to structural coverage. The empirical study also led to observation about the relationship between output and input that provide initial guidance to the design of a test generation approach based on output uniqueness alone.

The characteristics of a web application might affect the degree to which the approaches presented in this thesis could be successful in achieving high coverage and detecting fault. Specifically, the success of the presented approaches could depend on the richness of the output in terms of the embedding of input values in the output page (SWAT), the variety of values stored in session variables (SART), and the diversity of

the structure of the output (Output Uniqueness). Naturally, applications whose execution behaviour does not greatly depend on input values (for example, PHPSysInfo which only had four input parameters) might not benefit from a search based test data generation technique, such as the one presented in Chapter 3. An application that does not use session variables or a database will not benefit from the sequence regeneration technique presented in Chapter 4. However, such applications might not benefit from any sequence regeneration technique if no other state propagation methods are employed in the application. A web application where the output is not composed using HTML, for example, an application whose output is graphical or presented in XML would not be suitable for the use of the output definitions presented in Chapter 5 as a test selection criteria.

6.2 Future Work

Although the results obtained from SWAT and Dynamically Mined Value seeding **DMV** are promising, several other sources for seeding could be used. Client-side scripts perform validations on inputs and could therefore be a great resource in identifying pre-conditions on input values. In applications that use a database, some inputs might be coupled with database columns. An analysis of those relationships and an input specific database seeding approach could further improve effectiveness and efficiency.

Deciding input types remains the only process in SWAT that is not fully automated. The types of some inputs can be obtained by static analysis. However, the types of the reaming inputs have to be manually assigned by the tester. The static analysis phase could be enhanced to also analyse relationships between inputs and database columns. Moreover, since PHP variables are dynamically typed, the same input can be treated as an integer at one point and as a string at another, this manual effort might be eliminated by dynamically assigning types to inputs. The current implementation does not restrict DMV seeding of values to inputs of the same type specifically because of this dynamic typing feature of PHP. This could be extended to deduct information about types of inputs relevant to each target branch dynamically. However, the effects of such approach on effectiveness and efficiency have to be investigated.

SWAT introduced two types of dynamic seeding sources: Values mined from the output page and values collected during execution from predicates. Seeding from the output page is specific to web application or applications that have similar output characteristics (e.g., systems with graphical user interfaces). However, seeding from values collected from predicates could be applied to any type of system and might prove to show similar enhancements on effectiveness and efficiency. This can be especially expected for applications with string predicates, as was observed with web applications.

SART was applied to test suites that were generated to maximize branch coverage to generate sequences of requests that are more effective in detecting faults. However, the same principles apply to test suites generated using other techniques. Test suites generated from session data displayed increases in coverage and fault finding when randomly reordered without resetting the state [SGSP05b]. The approaches implemented in SART could be applied to session-data-based test suites to replace random reordering.

However, since session-data-based test suites are expected to contain redundancy, unlike test suites generated for branch coverage, the approach needs to be amended to limit the number of additional generated test sequences. Since VADU uses values stored in session variables and database SQL statements to generate Def-Use pairs, redundancy in test cases can lead to a large number of new sequences. To avoid producing a large number of Def-Use sequences for session variables and SQL statements that might be conceptually similar, an analysis to cluster values in these state variables could be performed. For example, two SQL statements that insert two different records will be treated as two different definition point in the current implementation. A quick analysis of the inserted records could decide dynamically whether those two insert statements are unlikely to add benefit and treat them as equivalent; for example when all values in the records are the same except for the record ID.

The current implementation of the approach does not attempt to minimize the number of new test sequences that cover all Def-Use pairs. An amendment to the algorithm that finds sequences that cover several Def-Use pairs at the same time could reduce the number of additional test sequences generated.

The application of dataflow analysis to database tables is not specific to web applications and can be applied to a variety of systems that use a database. Changes to

the implementation of VADU are minimal since database logs, which are available on any system that uses a database, are used to identify Def-Use points.

The combined SWAT and SART approach achieves high branch coverage for some of the applications studied. However, an investigation of branches reached but not covered reveals that there are some issues that require customized approaches to handle them. One such issue is the request method: In applications that use both `GET` and `POST` submit methods, the test generation approach can be misguided in which test inputs to mutate especially when the code that handles each submit method is intermingled. This can be observed in Timeclock and PHPBB2. An approach that can detect which submit method affects the target branch or an approach that generates test data for each submit method separately might increase both effectiveness and efficiency.

In other cases, the approaches fail to cover branches that do not depend on the user input. This is mainly observed in branches that depend on variables stored in configuration files, branches that depend on the database or branches that depend on the server's system. To cover branches that depend on configurations, a test generation approach can consider the variables in the configuration files as additional inputs to the application.

Some branches depend either directly or indirectly on values in the database. For example, in Schoolmate, when accessing the page that lists all users of the application, an SQL query of the `Users` table is performed followed by a branching statement to check if the query returned any results. If the `Users` table is empty, the `true` branch of that branching statement would never be covered. On the other hand, if the table contains records, the `false` branch would never be covered. In other cases, the dependence is not direct as the previous example. Branches at any point of the program can depend on values in the record(s) fetched from the database at preceding parts of the program. If no record in the database exists that satisfies the predicates in those branches, the branch cannot be covered. However, care has to be taken when populating the database artificially with values that help cover those branches: It might not be possible to populate the database with such values through the application's interface. Therefore, to avoid false positives, the database state should be changed through the application's interface.

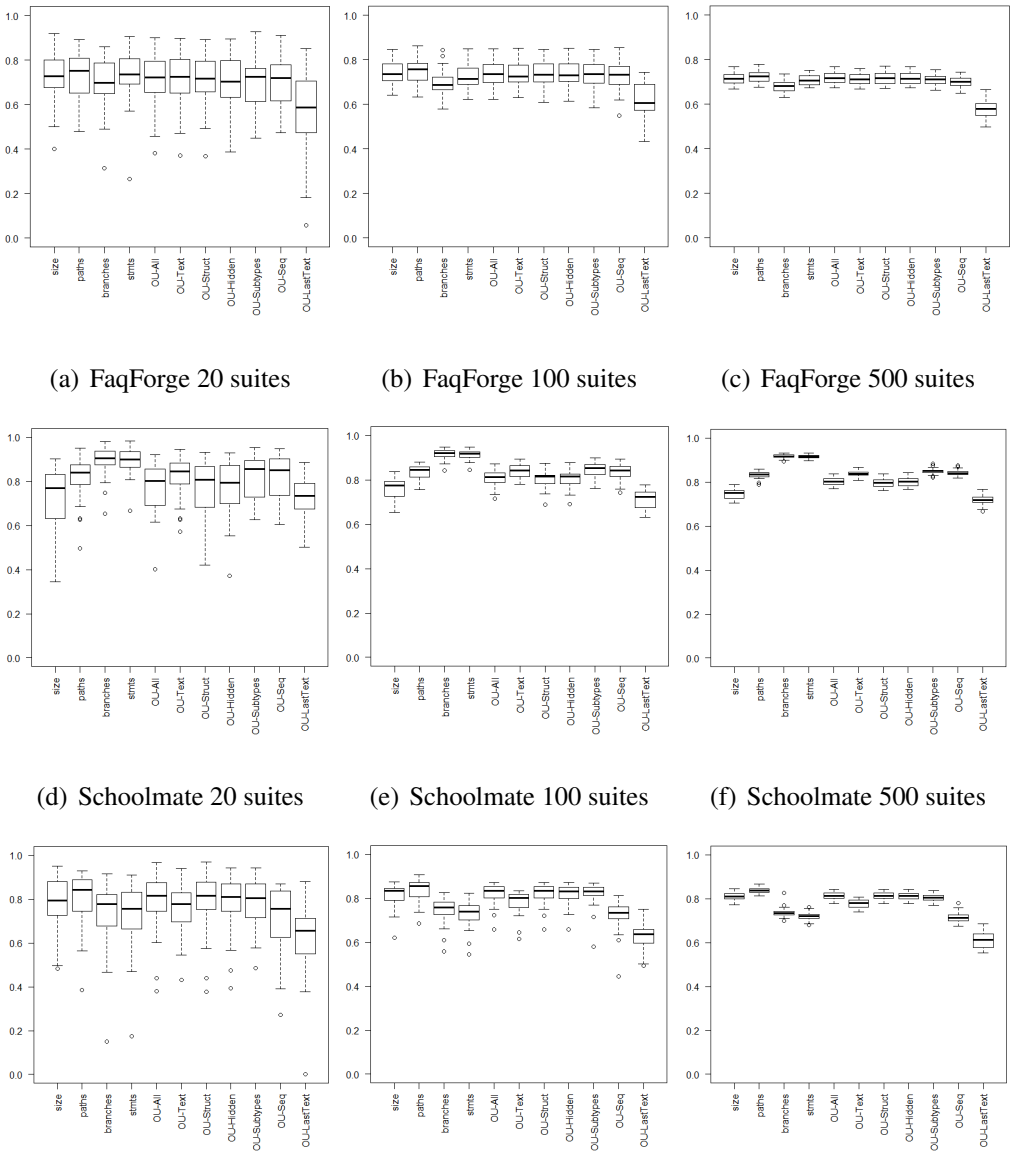
In special cases, branches depend on the system the application is installed on or the user's browser. For example, PHPSysInfo is an application that prints information about the server's system. The user choices are limited to output type (e.g., HTML or XML) or the output language. Branching statements in the code are mainly based on details about the system (e.g., operating system, memory usage). An effective testing approach for such application would be to install it on different types of systems.

The most exciting and immediate future work that arises from Chapter 5 is the design and construction of a test generation algorithm for web applications that is based on output uniqueness. The empirical study in Section 5.4 showed that a strong correlation exists between output uniqueness and structural coverage. A test generation tool that is solely based on output uniqueness has the benefit of being language independent. If such a tool proves successful in achieving high coverage and fault finding levels with manageable sized test suites, it can be applied to a variety of web applications developed using different technologies. Another benefit of such a tool is that it avoids the overhead of instrumentation and calculations needed for other techniques (fitness calculations in search based techniques or constraint solving in DSE).

The empirical study also showed that output uniqueness and structural coverage are complementary in consistency in finding faults. An approach that combines test generation for maximizing branch coverage, such as in SWAT, with an approach to maximize output diversity can produce test suites that are more effective than those generated to maximize branch coverage alone.

Appendix A

Effect of Set Size on Correlations

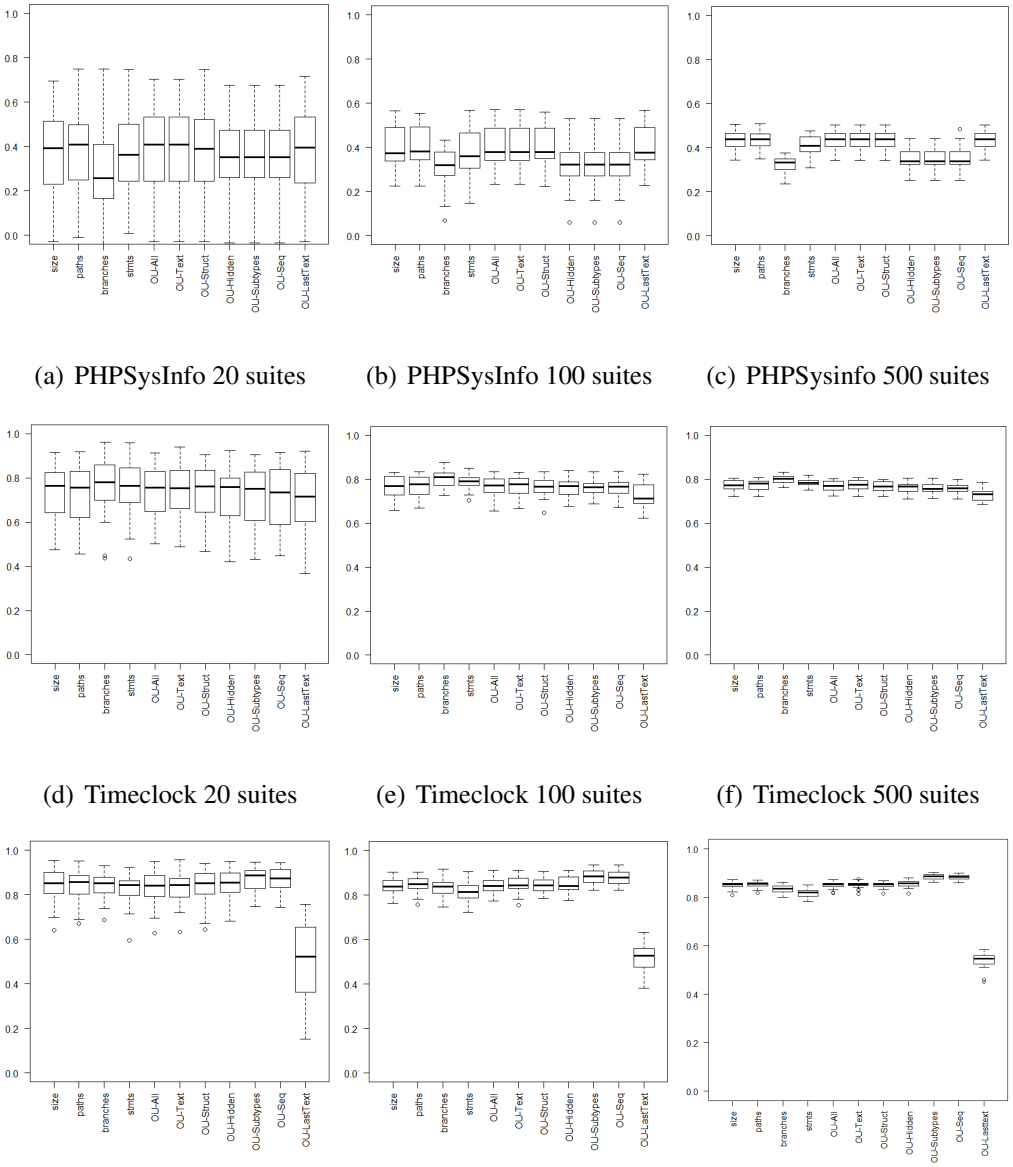


(a) FaqForge 20 suites (b) FaqForge 100 suites (c) FaqForge 500 suites

(d) Schoolmate 20 suites (e) Schoolmate 100 suites (f) Schoolmate 500 suites

(g) Webchess 20 suites (h) Webchess 100 suites (i) Webchess 500 suites

Figure A.1: Variations in Spearman's rank correlation coefficient between fault finding and test suite size, structural coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for FaqForge, Schoolmate and Webchess.

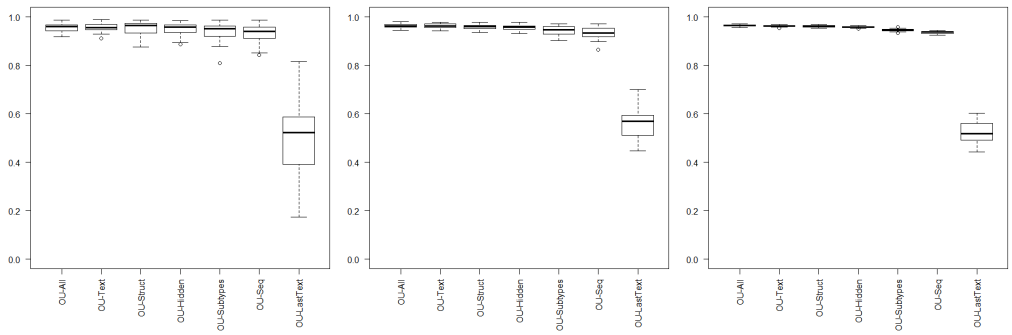


(a) PHPSysInfo 20 suites (b) PHPSysInfo 100 suites (c) PHPSysinfo 500 suites

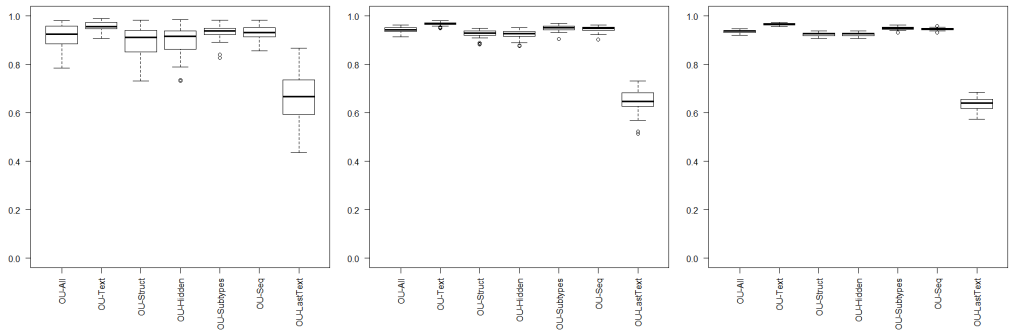
(d) Timeclock 20 suites (e) Timeclock 100 suites (f) Timeclock 500 suites

(g) PHPBB2 20 suites (h) PHPBB2 100 suites (i) PHPBB2 500 suites

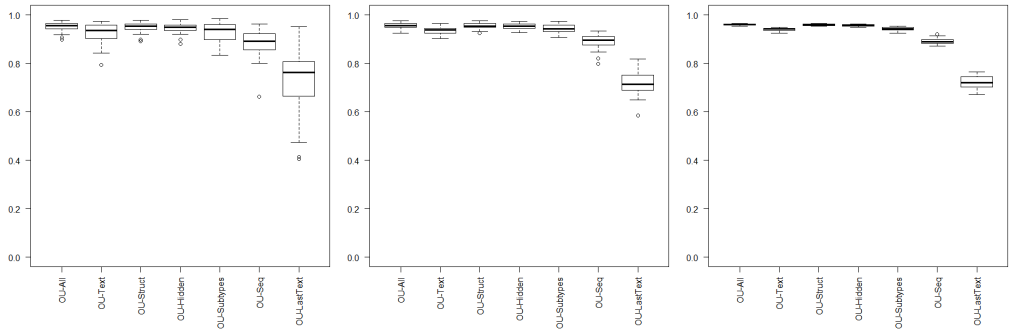
Figure A.2: Variations in Spearman's rank correlation coefficient between fault finding and test suite size, structural coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for PHPSysInfo, Timeclock and PHPBB2.



(a) FaqForge 20 suites (b) FaqForge 100 suites (c) FaqForge 500 suites

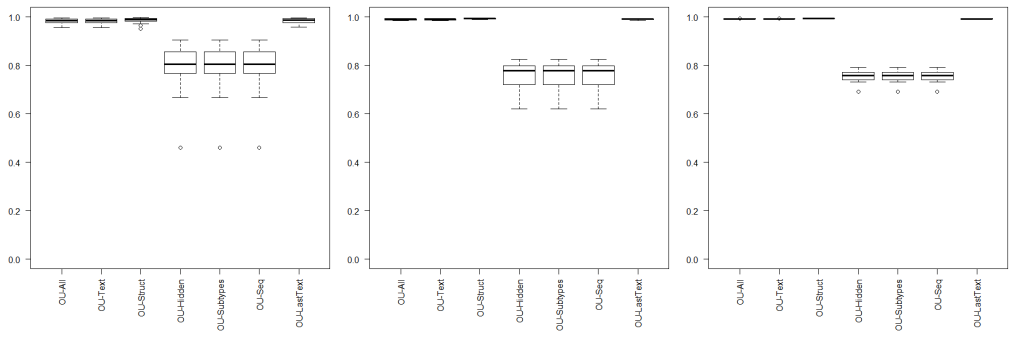


(d) Schoolmate 20 suites (e) Schoolmate 100 suites (f) Schoolmate 500 suites

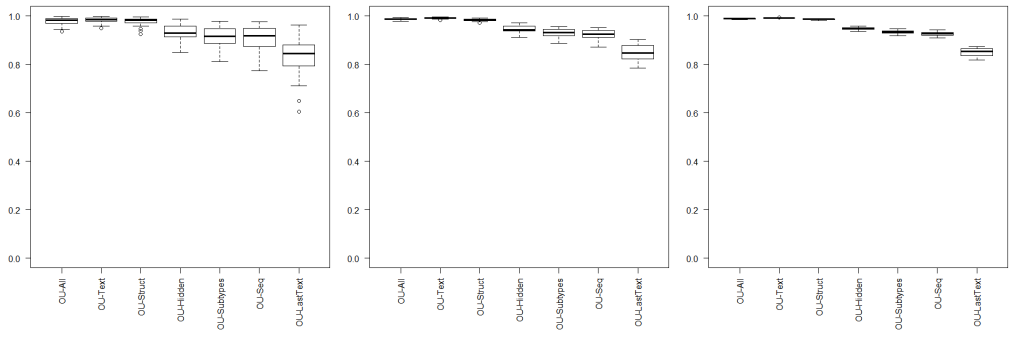


(g) Webchess 20 suites (h) Webchess 100 suites (i) Webchess 500 suites

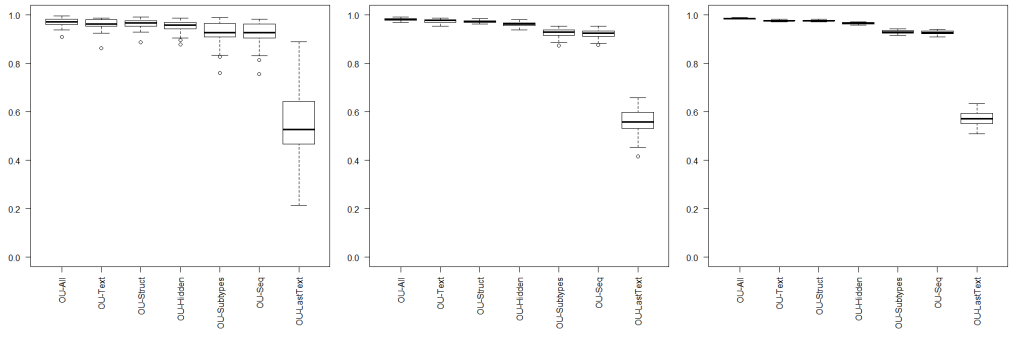
Figure A.3: Variations in Spearman's rank correlation coefficient between path coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for FaqForge, Schoolmate and Webchess.



(a) PHPSysInfo 20 suites (b) PHPSysInfo 100 suites (c) PHPSysinfo 500 suites

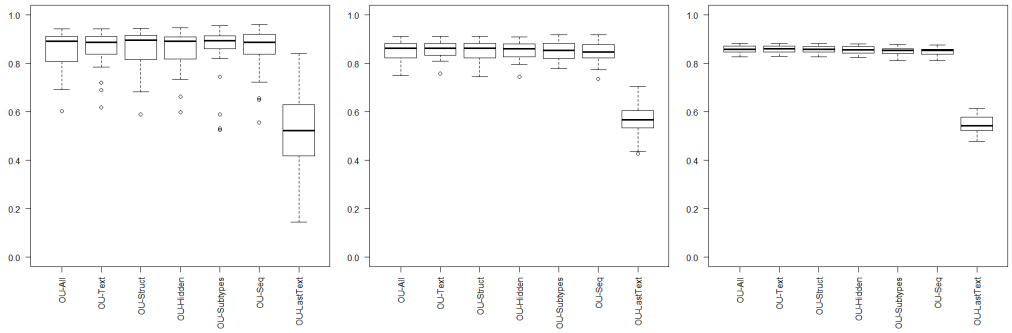


(d) Timeclock 20 suites (e) Timeclock 100 suites (f) Timeclock 500 suites

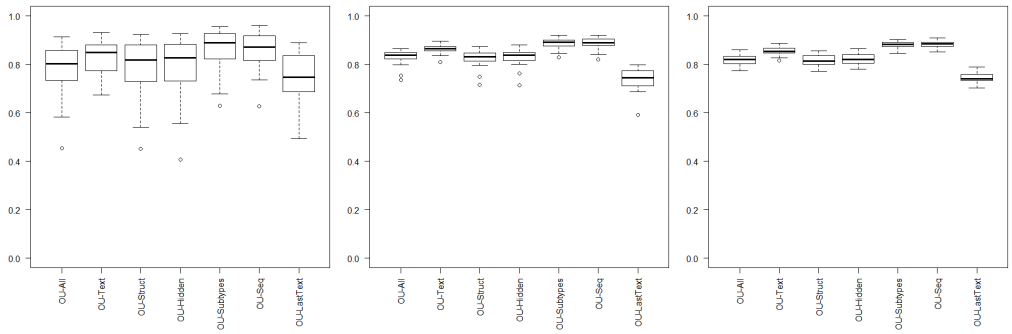


(g) PHPBB2 20 suites (h) PHPBB2 100 suites (i) PHPBB2 500 suites

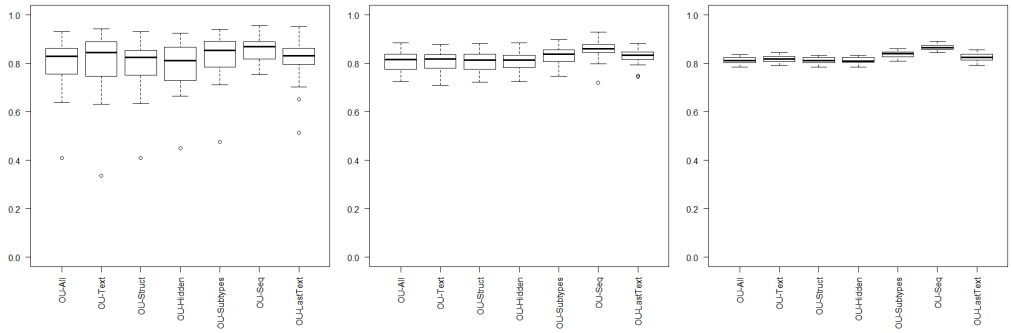
Figure A.4: Variations in Spearman's rank correlation coefficient between path coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for PHPSysInfo, Timeclock and PHPBB2.



(a) FaqForge 20 suites (b) FaqForge 100 suites (c) FaqForge 500 suites

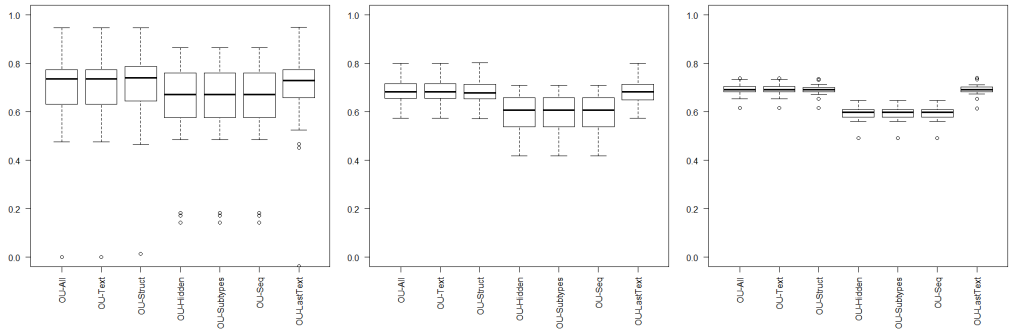


(d) Schoolmate 20 suites (e) Schoolmate 100 suites (f) Schoolmate 500 suites

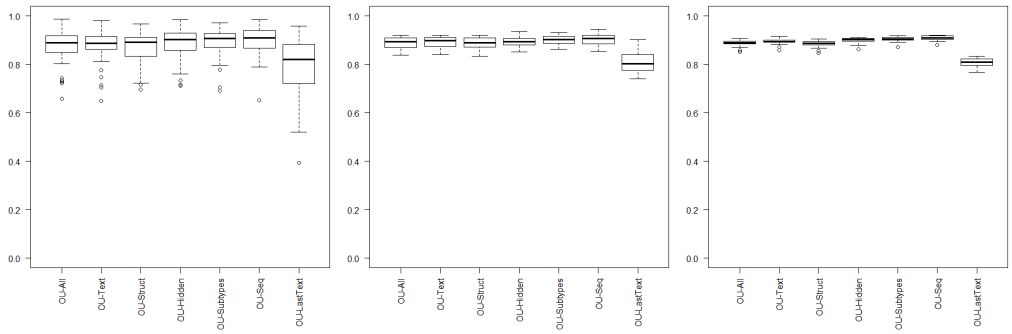


(g) Webchess 20 suites (h) Webchess 100 suites (i) Webchess 500 suites

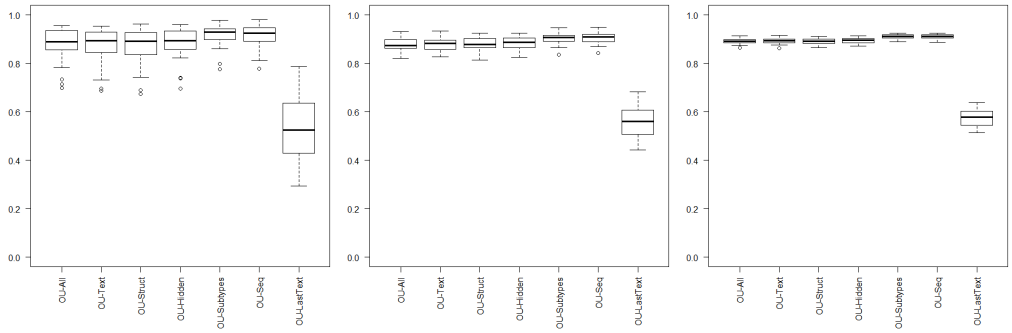
Figure A.5: Variations in Spearman's rank correlation coefficient between branch coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for FaqForge, Schoolmate and Webchess.



(a) PHPSysInfo 20 suites (b) PHPSysInfo 100 suites (c) PHPSysinfo 500 suites

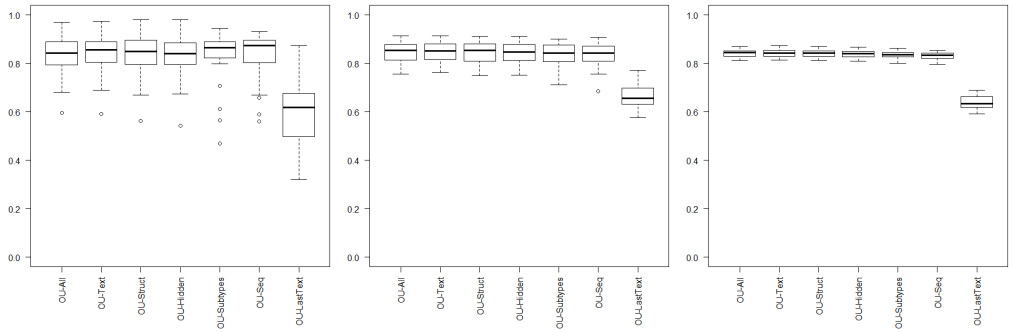


(d) Timeclock 20 suites (e) Timeclock 100 suites (f) Timeclock 500 suites

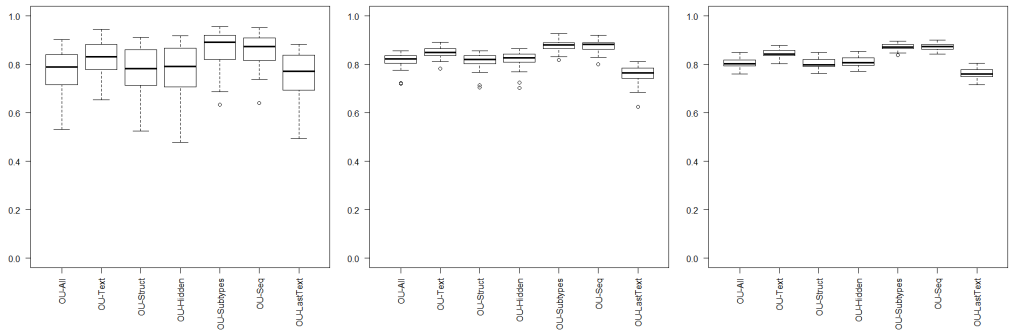


(g) PHPBB2 20 suites (h) PHPBB2 100 suites (i) PHPBB2 500 suites

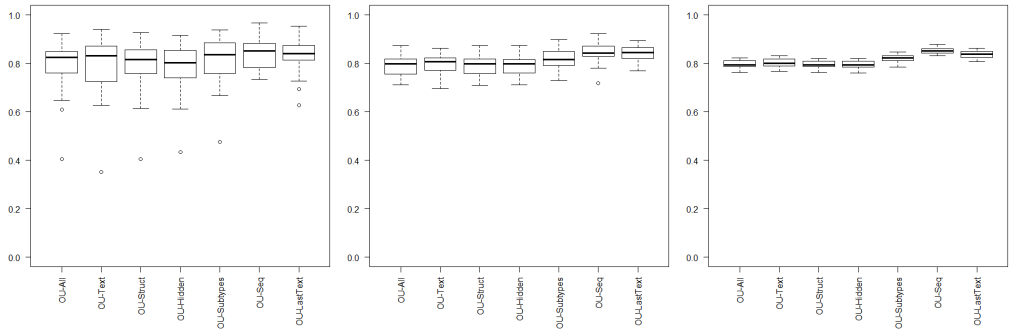
Figure A.6: Variations in Spearman's rank correlation coefficient between branch coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for PHPSysInfo, Timeclock and PHPBB2.



(a) FaqForge 20 suites (b) FaqForge 100 suites (c) FaqForge 500 suites

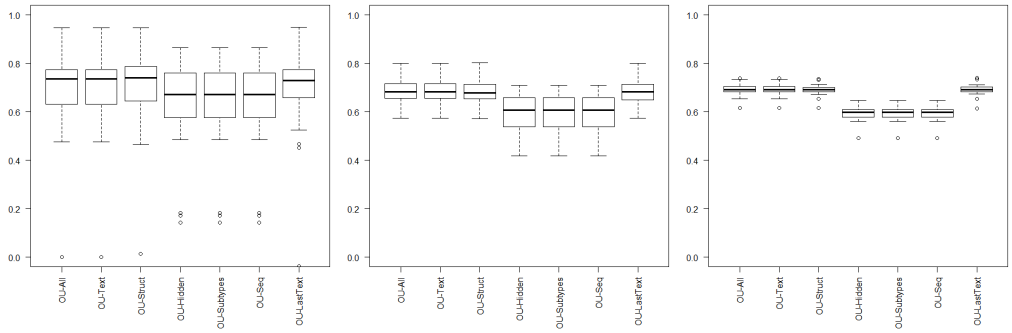


(d) Schoolmate 20 suites (e) Schoolmate 100 suites (f) Schoolmate 500 suites

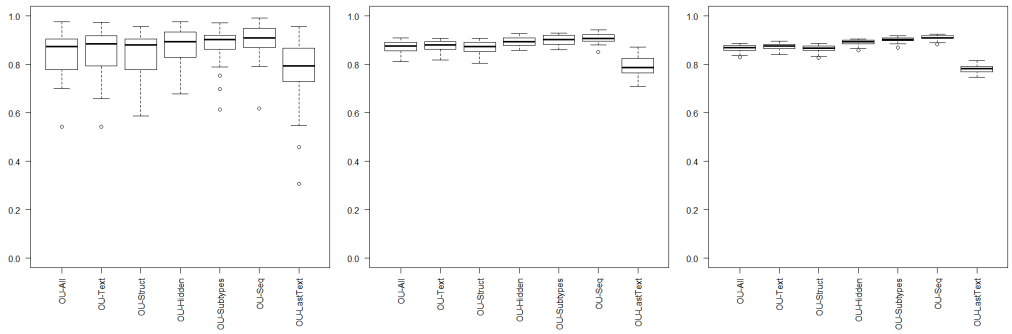


(g) Webchess 20 suites (h) Webchess 100 suites (i) Webchess 500 suites

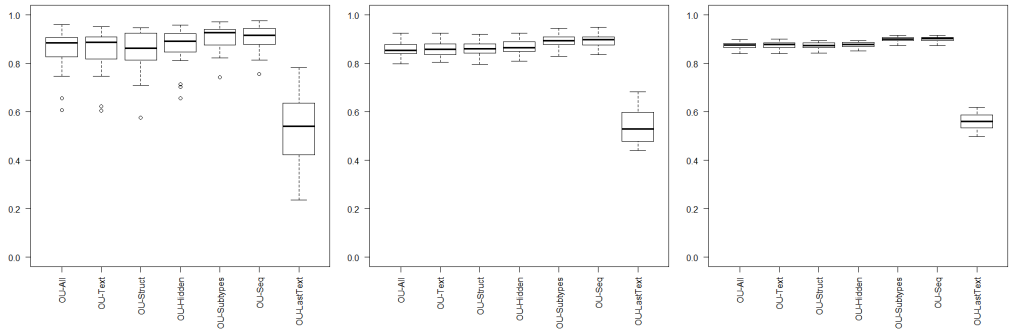
Figure A.7: Variations in Spearman's rank correlation coefficient between statement and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for FaqForge, Schoolmate and Webchess.



(a) PHPSysInfo 20 suites (b) PHPSysInfo 100 suites (c) PHPSysinfo 500 suites



(d) Timeclock 20 suites (e) Timeclock 100 suites (f) Timeclock 500 suites



(g) PHPBB2 20 suites (h) PHPBB2 100 suites (i) PHPBB2 500 suites

Figure A.8: Variations in Spearman's rank correlation coefficient between statement coverage and output uniqueness for test suites sets of sizes 20, 100 and 500 over 30 different experiments for PHPSysInfo, Timeclock and PHPBB2.

Bibliography

[AB06] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators: Research articles. *Software Testing, Verification and Reliability - UKTest 2005: The Third U.K. Workshop on Software Testing Research*, 16(3):175–203, September 2006.

[ABHPW10] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering (TSE)*, 36:742–762, November 2010.

[ABLN06] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)*, 32(8):608–624, August 2006.

[AC06] Enrique Alba and Francisco Chicano. Software testing with evolutionary strategies. In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques*, volume 3943 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin / Heidelberg, Crete, Greece, 2006.

[AC08] Enrique Alba and Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers and Operations Research*, 35:3161–3183, October 2008.

[ADTP10] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th*

International Symposium on Software Testing and Analysis (ISSTA '10), pages 49–60. ACM, 2010.

[AH08] Nadia Alshahwan and Mark Harman. Automated session data repair for web application regression testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 298–307. IEEE Computer Society, 2008.

[AH11] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pages 3–12, 2011.

[AH12a] Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity (NIER track). In *Proceedings of the 34rd International Conference on Software Engineering - New Ideas and Emerging Results Track (ICSE NIER '12)*, 2012. to appear.

[AH12b] Nadia Alshahwan and Mark Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *the 21st International Symposium on Software Testing and Analysis (ISSTA '12)*, 2012. to appear.

[AKD+08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA '08)*, pages 261–272. ACM, 2008.

[AKD+10] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering (TSE)*, 36:474–494, July 2010.

[AOA05] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4:326–345, 2005.

[Arc10] Andrea Arcuri. Longer is better: On the role of test sequence length in software testing. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 469–478. IEEE Computer Society, 2010.

[AS03] Rolph E. Anderson and Srini S. Srinivasan. E-satisfaction and e-loyalty: A contingency framework. *Journal of Psychology and Marketing*, 20(2):123–138, 2003.

[BB10] Eric Bouwers and Martin Bravenboer. PHP-front: Static analysis for PHP. http://strategoxt.org/PHP/PhpFront, 2010.

[BFG02] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In *Proceedings of 11th International Conference on World Wide Web (WWW '02)*, Honolulu, HI, USA, 2002.

[BGFT09] Raquel Blanco, José García-Fanjul, and Javier Tuya. A first approach to test case generation for BPEL compositions of web services using scatter search. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*, pages 131–140. IEEE Computer Society, 2009.

[BHH12] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing & verification in service-oriented architecture: A survey. *Software Testing, Verification and Reliability (STVR)*, 2012. to appear.

[BKMD09] Thomas G. Brashear, Vishal Kashyap, Michael D. Musante, and Naveen Donthu. A profile of the internet shopper: Evidence from six countries. *The Journal of Marketing Theory and Practice*, 17(3):267–282, 2009.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

[BMT05] Carlo Bellettini, Alessandro Marchetto, and Andrea Trentini. TestUml: user-metrics driven web applications testing. In *Proceedings of the ACM*

Symposium on Applied Computing (SAC '05), pages 1694–1698. ACM, 2005.

[CC99] Man-Yee Chan and Shing-Chi Cheung. Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS '99)*, pages 363–374, 1999.

[CDF+04] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An AGENDA for testing relational database applications: Research articles. *Software Testing, Verification and Reliability (STVR)*, 14:17–44, March 2004.

[CUR+02] Wei Chen, Roland H. Untch, Gregg Rothermel, Sebastian Elbaum, and Jeffery Von Ronne. Can fault-exposure-potential estimates improve the fault detection abilities of test suites? *Software Testing, Verification and Reliability (STVR)*, 12(4):197–218, December 2002.

[DA99] Naveen Donthu and Garcia Adriana. The internet shopper. *Journal of Advertising Research*, 39(3):52–58, 1999.

[DEW97] Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the 1st International Conference on Autonomous Agents (AGENTS '97)*, pages 39–48. ACM, 1997.

[DFW04] Yuetang Deng, Phyllis Frankl, and Jiong Wang. Testing web database applications. *SIGSOFT Software Engineering Notes*, 29:1–10, September 2004.

[DLDB06] Giuseppe Di Lucca, Damiano Distante, and Mario Luca Bernardi. Recovering conceptual models from web applications. In *Proceedings of the 24th Annual ACM International Conference on Design of Communication (SIGDOC '06)*, pages 113–120. ACM, 2006.

[DLDP03] Giuseppe Di Lucca and Massimiliano Di Penta. Considering browser interaction in web application testing. In *Proceedings of the 5th IEEE*

International Workshop on Web Site Evolution (WSE '03), pages 74–81. IEEE Computer Society, 2003.

[DLDPF02] Giuseppe Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC '02)*, pages 481–486. IEEE Computer Society, 2002.

[DLF06] Giuseppe Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48:1172–1186, December 2006.

[DLFF02] Giuseppe Di Lucca, Anna Rita Fasolino, and Francesco Faralli. Testing web applications. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 310–319. IEEE Computer Society, 2002.

[DLFP+02] Giuseppe Di Lucca, Anna Rita Fasolino, F. Pace, Porfirio Tramontana, and Ugo de Carlini. WARE: A tool for the reverse engineering of web applications. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR '02)*, pages 241–250. IEEE Computer Society, 2002.

[DLFT04] Giuseppe Di Lucca, Anna Rita Fasolino, and Porfirio Tramontana. Reverse engineering web applications: the WARE approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):71–101, January 2004.

[DW10] Kinga Dobolyi and Westley Weimer. Modeling consumer-perceived web application fault severities for testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*, pages 97–106. ACM, 2010.

[ECIR06] Sebastian Elbaum, Kalyan-Ram Chilakamarri, Marc Fisher II, and Gregg Rothermel. Web application characterization through directed re-

quests. In *Proceedings of the 4th International Workshop on Dynamic Systems Analysis (WODA '06)*, pages 49–56. ACM, 2006.

[EKR03] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 49–59. IEEE Computer Society, 2003.

[EM07] Cyntrica Eaton and Atif M. Memon. An empirical approach to evaluating web application compliance across diverse client platform configurations. *International Journal of Web Engineering and Technology*, 3(3):227–253, January 2007.

[EMS07] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 151–162. ACM, 2007.

[ERKI05] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering (TSE)*, 31(3):187–202, March 2005.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, January 1996.

[FW93] Phyllis G Frankl and Stewart N Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[Ger00] Paul Gerrard. Risk-based e-business testing, part 1 risks and test strategy. http://www.gerrardconsulting.com, 2000.

[GG09] Yuanyan Gu and Yujia Ge. Search-based performance testing of applications with composite services. In *Proceedings of the 2009 International*

Conference on Web Information Systems and Mining (WISM '09), pages 320–324. IEEE Computer Society, 2009.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223. ACM, 2005.

[Glo90] Fred Glover. Tabu search: A tutorial. *Interfaces 20*, pages 74–94, 1990.

[GN97] Matthew J. Gallagher and V. Lakshmi Narasimhan. ADTEST: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering (TSE)*, 23(8):473–484, August 1997.

[Gol08] Russell Gold. HttpUnit. http://httpunit.sourceforge.net, 2008.

[GRT06] Christian Girardi, Filippo Ricca, and Paolo Tonella. Web crawlers compared. *International Journal of Web Information Systems*, 2(2):85–94, 2006.

[HAO09] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09)*, pages 285–296. ACM, 2009.

[Har07] Mark Harman. The current state and future of search based software engineering. In *Proceedings of the 29th International Conference on Software Engineering - Future of Software Engineering (FOSE '07) Track*, pages 342–357. IEEE Computer Society, 2007.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*, pages 191–200. IEEE Computer Society Press, 1994.

[HHH+04] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, Andre Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering (TSE)*, 30:3–16, January 2004.

[HJ01] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[HM02] Edward Hieatt and Robert Mee. Going faster: Testing the web application. *IEEE Software*, 19(2):60–65, March/April 2002.

[HM07] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 73–83. ACM, 2007.

[HM10] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, March 2010.

[HN99] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web Journal*, 2(4):219–229, 1999.

[HO06] William G. J. Halfond and Alessandro Orso. Command-Form coverage for testing database applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pages 69–80. IEEE Computer Society, 2006.

[HO07] William G. J. Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 145–154. ACM, 2007.

[HO08] William G. J. Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the 16th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16), pages 181–191. ACM, 2008.

[HR94] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. *SIGSOFT Software Engineering Notes*, 19:154–163, December 1994.

[IER07] Marc Fisher II, Sebastian Elbaum, and Gregg Rothermel. Dynamic characterization of web application interfaces. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE'07)*, pages 260–275. Springer Berlin / Heidelberg, 2007.

[Jav03] JavaCoding. JSpider. http://j-spider.sourceforge.net, 2003.

[JHHF08] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM '08)*, pages 307–316. IEEE Computer Society, 2008.

[JL02] Xiaoping Jia and Hongming Liu. Rigorous and automatic testing of web applications. In *Proceeding of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 280–285, 2002.

[JSE96] Bryan F. Jones, Harmen Sthamer, and David E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11:299–306, September 1996.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering (TSE)*, 16(8):870–879, August 1990.

[KT01] Chaitanya Kallepalli and Jeff Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering (TSE)*, 27:1023–1036, November 2001.

[LESY03] Stephen W. Liddle, David W. Embley, Del T. Scott, and Sai Ho Yau. Extracting data behind web forms. *Advanced Conceptual Modeling Techniques*, 2784:402–413, 2003.

[LHM07] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, pages 1098–1105. ACM, 2007.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering (TSE)*, 9:347–354, May 1983.

[LK04] Daniel R. Licata and Shriram Krishnamurthi. Verifying interactive web programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*, pages 164–173. IEEE Computer Society, 2004.

[LKHH00a] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. Object-based data flow testing of web applications. In *Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS'00)*, pages 7–16. IEEE Computer Society, 2000.

[LKHH00b] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. Structural testing of web applications. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE '00)*, pages 84–96. IEEE Computer Society, 2000.

[LMH09] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART '09)*, pages 95–104. IEEE Computer Society, 2009.

[LMH10] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83:2379–2391, December 2010.

[LPC09] Xingmin Luo, Fan Ping, and Mei-Hwa Chen. Clustering and tailoring user session data for testing web applications. In *Proceedings of the 2nd*

International Conference on Software Testing Verification and Validation (ICST '09), pages 336–345. IEEE Computer Society, 2009.

[LYE02] Stephen W. Liddle, Sai Ho Yau, and David W. Embley. On the automatic extraction of data from the hidden web. In *Revised Papers from the HUMACS, DASWIS, ECOMO, and DAMA on ER 2001 Workshops*, pages 212–226. Springer-Verlag, 2002.

[LYE03] Stephen W. Liddle, Sai Ho Yau, and David W. Embley. Extracting data behind web forms. In *Advanced Conceptual Modeling Techniques*, volume 2784/2003, pages 402–413. Springer Berlin / Heidelberg, October 2003.

[MB98] Robert C. Miller and Krishna Bharat. Sphinx: a framework for creating personal, site-specific web crawlers. *Computer Networks and ISDN Systems*, 30(1-7):119–130, April 1998.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability (STVR)*, 14(2):105–156, June 2004.

[Mes03] Gerard Meszaros. Agile regression testing using record & playback. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 353–360. ACM, 2003.

[MHBT06] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to search based test data generation. In *Proceedings of the 15th International Symposium on Software Testing and Analysis (ISSTA '06)*, pages 13–24. ACM, 2006.

[Min05] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web (WWW '05)*, pages 432–441. ACM, 2005.

[MMS01] Christoph C. Michael, Gary E. McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering (TSE)*, 27:1085–1110, December 2001.

[MOP02] Vincenzo Martena, Alessandro Orso, and Mauro Pezzé. Interclass testing of object oriented software. In *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS '02)*, pages 135–144. IEEE Computer Society, 2002.

[MP11] Ali Mesbah and Mukul R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 561–570. ACM, 2011.

[MRT08] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. A case study-based comparison of web testing techniques applied to AJAX web applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):477–492, October 2008.

[MS76] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering (TSE)*, 2(3):223–226, May 1976.

[MSP01] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9 '01)*, pages 256–267. ACM, 2001.

[MT09] Alessandro Marchetto and Paolo Tonella. Search-based testing of AJAX web applications. In *Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09)*, pages 3–12. IEEE Computer Society, 2009.

[MT11] Alessandro Marchetto and Paolo Tonella. Using search-based algorithms for AJAX event sequence generation during testing. *Empirical Software Engineering*, 16:103–140, February 2011.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of AJAX web applications. In *Proceedings of the 1st International Conference on Software Testing Verification and Validation (ICST '08)*, pages 121–130. IEEE Computer Society, 2008.

[NA09] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09)*, pages 57–68. ACM, 2009.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.

[OP09] Oliverbock and Pixel. MaxQ. http://maxq.tigris.org, 2009.

[OWDH04a] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '04)*, pages 187–197. IEEE Computer Society, 2004.

[OWDH04b] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Web application bypass testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02 (COMPSAC '04)*, pages 106–109. IEEE Computer Society, 2004.

[OWO08] Jeff Offutt, Qingxiang Wang, and Joann Ordille. An industrial case study of bypass testing on web applications. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 465–474. IEEE Computer Society, 2008.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07)*, pages 815–816. ACM, 2007.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability (STVR)*, 9(4):263–282, December 1999.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .NET with feedback-directed random testing. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA '08)*, pages 87–96. ACM, 2008.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 75–84. IEEE Computer Society, 2007.

[PN05] Soila Pertet and Priya Narasimhan. Causes of failures in web applications. Technical report, Carnegie Mellon University, 2005.

[QMZ07] Zhongsheng Qian, Huaikou Miao, and Hongwei Zeng. A practical web testing model for web application testing. In *Proceedings of the 3rd International IEEE Conference on Signal-Image Technologies and Internet-Based System (SITIS '07)*, pages 434–441. IEEE Computer Society, 2007.

[RCVO10] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*, pages 1–10. IEEE Computer Society, 2010.

[Ret09] Internet Retailer. Top 500 e-retailers take a bigger bite of the pie. http://www.internetretailer.com, June 2009.

[Ret12] Derick Rethans. Xdebug. http://xdebug.org, 2012.

[RGM01] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 129–138. Morgan Kaufmann Publishers Inc., 2001.

[Rop97] Marc Roper. Computer aided software testing using genetic algorithms. In *Proceedings of the 10th International Software Quality Week*, 1997.

[RT00] Filippo Ricca and Paolo Tonella. Web site analysis: Structure and evolution. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM '00)*, pages 76–87. IEEE Computer Society, 2000.

[RT01a] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 25–34. IEEE Computer Society, 2001.

[RT01b] Filippo Ricca and Paolo Tonella. Building a tool for the analysis and testing of web applications: Problems and solutions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, pages 373–388. Springer-Verlag, 2001.

[RT01c] Filippo Ricca and Paolo Tonella. Understanding and restructuring web sites with reweb. *IEEE MultiMedia*, 8:40–51, April 2001.

[RT02] Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Annals of Software Engineering*, 14:93–114, December 2002.

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the 6th International Conference on Software Engineering (ICSE'82)*, pages 272–278. IEEE Computer Society Press, 1982.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering (TSE)*, 11:367–375, April 1985.

[SBV+08] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 141–150. IEEE Computer Society, 2008.

[SCA+08] Raúl A. Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 218–227. IEEE, 2008.

[SGSP05a] Sreedevi Sampath, Emily Gibson, Sara Sprenkle, and Lori Pollock. Coverage criteria for testing web applications. Technical Report 2005-17, University of Delaware, 2005.

[SGSP05b] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 253–262. ACM, 2005.

[Sim12] SimpleTest. Unit testing for PHP. http://www.simpletest.org, 2012.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13 '05)*, pages 263–272. ACM, 2005.

[SMSP04] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. Composing a framework to automate testing of operational web-based software. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 104–113. IEEE Computer Society, 2004.

[SSG+05a] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter. Analyzing clusters of web application user sessions. In *Proceedings of the 3rd International Workshop on Dynamic Systems Analysis (WODA '05)*, pages 1–7. ACM, 2005.

[SSG+05b] Sara Sprenkle, Sreedevi Sampath, Emily Gibson, Lori Pollock, and Amie Souter. An empirical comparison of test suite reduction techniques

for user-session-based testing of web applications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 587–596. IEEE Computer Society, 2005.

[SSP04] Sreedevi Sampath, Amie L. Souter, and Lori Pollock. Towards defining and exploiting similarities in web application use cases through user session analysis. In *Proceedings of the International Workshop on Dynamic Systems Analysis (WODA '04)*, pages 17–24. IEEE, 2004.

[Sta11] Internet World Stats. World internet users and population stats. http://www.internetworldstats.com, December 2011.

[TCM98] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test case generation: The optimisation-based approach. In *Proceedings of IFIP International Workshop on Dependable Computing and its Applications (DCIA '98)*, pages 169–180, 1998.

[TCMM98] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE '98)*, pages 285–288. IEEE Computer Society, 1998.

[Tec10] TechCrunsh. Forrester forecast: Online retail sales will grow to $250 billion by 2014. http://techcrunch.com, March 2010.

[TIO12] TIOBE Software. TIOBE programming community index for april 2012. http://www.tiobe.com/tpci.htm, April 2012.

[TM08] Andrew Tappenden and James Miller. A three-tiered testing strategy for cookies. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 131–140. IEEE Computer Society, 2008.

[Too01] LogiTest Web Application Testing Tools. LogiTest. http://logitest.sourceforge.net, 2001.

[TR04a] Paolo Tonella and Filippo Ricca. A 2-layer model for the white-box testing of web applications. In *Proceedings of the 6th IEEE International Workshop on Web Site Evolution (WSE '04)*, pages 11–19. IEEE Computer Society, 2004.

[TR04b] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. *Software Maintenance and Evolution: Research and Practice*, 16:103–127, January 2004.

[Tra00] Nigel Tracey. *A search-based automated test-data generation framework for safety critical software*. PhD thesis, University of York, York, UK, 2000.

[Tür11] Sven Türpe. Search-based application security testing: Towards a structured search space. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*, pages 198–201. IEEE Computer Society, 2011.

[TZPC05] Wei-Tek Tsai, Dawei Zhang, Raymond Paul, and Yinong Chen. Stochastic voting algorithms for web services group testing. In *Proceedings of the 5th International Conference on Quality Software (QSIC '05)*, pages 99–108. IEEE Computer Society, 2005.

[Wat95] Alison Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the 4th International Software Quality Week*, pages 300–309, 1995.

[WBP02] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '02)*, pages 1233–1240. Morgan Kaufmann Publishers Inc., 2002.

[WO02] Ye Wu and Jeff Offutt. Modeling and testing web-based applications. Technical Report ISE-TR-02-08, George Mason University, 2002.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 32–41. ACM, 2007.

[WYC+08] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA '08)*, pages 249–260. ACM, 2008.

[YH10] Shin Yoo and Mark Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability (STVR)*, 2010. to appear.

[YHW+02] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang, William, and C. Chu. Constructing an object-oriented architecture for web application testing. *Journal of Information Science and Engineering*, 18:59–84, January 2002.

[YHWC99] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang, and William C. Chu. An object-oriented architecture supporting web application testing. In *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC '99)*, pages 122–127. IEEE Computer Society, 1999.

[YM07] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 396–405. IEEE Computer Society, 2007.

[YM10] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering (TSE)*, 36(1):81–95, January 2010.

[Zhu95] Hong Zhu. Axiomatic assessment of control flow-based software test adequacy criteria. *Software Engineering Journal*, 10(5):194–204, September 1995.

[ZLM10] Ruilian Zhao, Michael R. Lyu, and Yinghua Min. Automatic string test data generation for detecting domain errors. *Software Testing, Verification and Reliability (STVR)*, 20:209–236, September 2010.