State Aware Test Case Regeneration for Improving Web Application Test Suite Coverage and Fault Detection

Nadia Alshahwan and Mark Harman
CREST Centre
University College London, Malet Place, London, WC1E 6BT, U.K.
{nadia.alshahwan.10,mark.harman}@ucl.ac.uk

ABSTRACT

This paper introduces two test cases regeneration approaches for web applications, one uses standard Def-Use testing but for state variables, the other uses a novel value-aware dataflow approach. Our overall approach is to combine requests from a test suite to form client-side request sequences, based on dataflow analysis of server-side session variables and database tables. We implemented our approach as a tool SART (State Aware Regeneration Tool) and used it to evaluate our proposed approaches on 4 real world web applications. Our results show that for all 4 applications, both server-side coverage and fault detection were statistically significantly improved. Even on relatively high quality test suites our algorithms improve average coverage by 14.74% and fault detection by 9.19%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Testing, Web applications, Regeneration, State

1. INTRODUCTION

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 [27] 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. [26] 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 re-ordered request sequences. However, exhaustive exploration of all possible such re-orderings will be prohibitively expensive, even for small test suites. Therefore, to get additional value from request orderings we need to develop intelligent algorithms 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, we seek 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, we seek request sequences that cause the server to exercise the server-side web application state.

We introduce two techniques for generating new sequences of HTTP requests from an existing test suite of requests. Both our techniques are 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.

We introduce a 'standard' Def-Use (DU) approach that seeks to ensure that each definition flows to each use. We also introduce 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. We implement each of these techniques in a tool SART (State Aware Regeneration Tool) and use this to experiment with our two approaches.

We report the results of an empirical study on four real world web applications for our state-based and value-aware dataflow techniques and also, as a baseline comparator, for random recombination. In all experiments we start with high quality test suites drawn from those whose cases alone, when executed individually, already achieve good coverage and fault detection. We do this in order to test the ability of our recombination to add value; clearly were we to start with a low-coverage request set, then any recombination can be excepted to produce additional coverage and may thereby be sufficiently fortunate to reveal additional faults. However, we show that our approach (and in particular the value-aware approach) can statistically significantly improve coverage and fault detection for relatively high quality test suites.

The primary contributions of the paper 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 two approaches on 4 real-world web applications that confirms the effectiveness and efficiency of these approaches. 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 paper is organized as follows: Section 2 provides a background concerning web application state. Section 3 introduces the two proposed approaches, whilst Section 4 introduces our tool called SART that implements the approaches. Section 5 presents the evaluation together with a discussion of the results. Section 6 presents related work and Section 7 concludes.

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.

In this paper we focus on PHP server-side code for concreteness, but the techniques we introduce can be applied to other server-side languages.

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].

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.

3. APPROACH

For our 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. We seek 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 we regenerate can come from any existing approach [2, 5, 15, 24, 28].

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

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, we propose an approach that generates test sequences by combining HTTP requests that define and use the server-side state. Our approach shares the same principles embodied in existing dataflow testing approaches [18, 22]. However, the Def-Use approach we propose 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. We use statement locations to identify distinct definition and use points. We employ dataflow testing principles and augment them to generate test sequences that are more likely to enhance the effectiveness of the original test suites in both coverage and fault finding.

Our 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. We then append 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 our focus on session variables and database tables and the need to execute a sequence of requests to activate server-side definitions and uses.

We also introduce a value-aware Def-Use testing approach that is more fine grained; 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 we shall show in our empirical results, for web applications this approach is not only feasible, but it produces a significant improvement in coverage and fault detection.

In the reminder of this section we describe our state-based DU and value-aware DU approaches in more detail.

3.1 State-based DU

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

We first perform a simple static analysis 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, we can only execute a Def-Use pair by issuing a sequence of requests from the client-side. For session functions, we classify them 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, we consider the session array 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.

To identify database tables' definitions and uses, we need 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. We collect locations of calls to the function `mysql_query` in the code and the SQL statements used at each call 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 we use 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 string and the user-provided input and dynamically computed variables. In these cases we 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 for the purpose of this paper it is sufficient for the approximated string to 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 we can automatically extract the operation `UPDATE` and the affected table `FaqPage` and conclude that this is a definition of table `FaqPage`.

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 1 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: $TS' = \phi$
 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:          $newSeq$ = ($deftest$,$usetestid$)
 8:          $TS' = TS' \cup newSeq$
 9:       end for
10:    end for
11: end for
12: return $TS'$
```

Algorithm 1 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, a HTTP request that executes that location is used to construct a new test sequence (Lines 5-8). The output of the algorithm is a set of new test sequences (TS′).

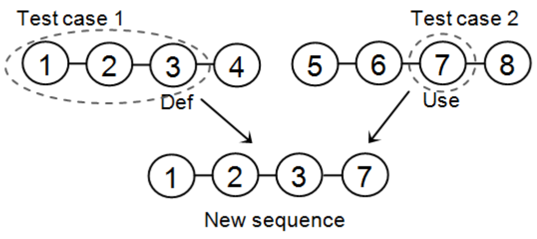


Figure 1: DU Sequence construction technique: The sequence that contains the definition HTTP request is truncated after the request and combined with the use HTTP request to form the new sequence.

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 exist 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). Figure 1 illustrates how sequences are generated.

In the reminder of the paper we will refer to this approach as State-based DU (or DU for short).

3.2 Value-Aware DU

The State-based DU approach, since it is based on a traditional DU testing approach, does not take into account the specific values session variables hold nor the specific database statements executed when database invocations are created dynamically.

Since session variables are used to hold information needed by the server to process the user's next request, we expect 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.

We therefore develop a dataflow testing approach that takes into account these different values and effectively treats each distinct value as a separate definition. We believe that this is 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 our evaluation.

While performing the dynamic analysis of the original test suite, for each HTTP request we also note 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 2 describes our 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 8). 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, we expect that different string values used as SQL calls may also denote different application behaviours.

We monitor how each test case interacts with the database by collecting every concrete SQL statement it executes. We use this information to construct the new sequences. Using this dynamic analysis we also avoid the limitations of static analyses in the approximation of dynamic strings, since at run-time, we know the value of the SQL statement string.

Algorithm 3 describes our 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 the static analysis results. For every database table, all distinct SQL statements (including those generated dynamically) that alter the database are retrieved (Line 4).

Algorithm 2 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: $TS' = \phi$
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:             $usetestid$ = getusetestcase($TS$,$sv$,$useloc$)
8:             $newSeq = (deftest, usetestid)$
9:             $TS' = TS' \cup newSeq$
10:         end for
11:      end for
12:   end for
13: end for
14: return $TS'$
```

A test case that executes the definition statement is then paired with all HTTP requests that use the same table. Because we append *all* uses to the same definition, we reduce the computational effort involved in testing multiple uses. The definition part will only need to be executed once instead of being repeated for every use. However, we insure 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 2 illustrates how VADU database table sequences are generated.

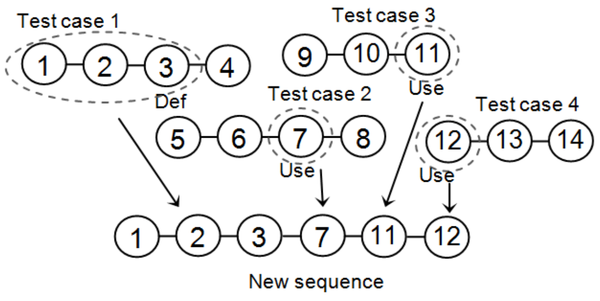


Figure 2: VADU Sequence construction technique: The sequence that contains the definition HTTP request is truncated after the request and combined with all use HTTP request to form the new sequence.

The output of the algorithm is a set of new test sequences (TS$''$). The union of TS$'$ from Algorithm 2 and TS$''$ from Algorithm 3 is the overall output test sequence set for the VADU approach.

In the reminder of the paper we will refer to this approach as VADU.

Algorithm 3 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: $TS'' = \phi$
 2: for all $tab$ in $Tables$ do
 3:   $newSeq = \phi$
 4:   for all $stmt$ in $SA$ do
 5:     $deftest$ = getdeftestcase($TS$,$stmt$)
 6:     $newSeq = newSeq \cup deftest$
 7:     for all $stmt$ in $SU$ do
 8:       $usetestids$ = getusetestcases($TS$,$stmt$)
 9:       for all $testid$ in $usetestids$ do
10:         $newSeq = newSeq \cup testid$
11:       end for
12:     end for
13:     $TS'' = TS'' \cup newSeq$
14:   end for
15: end for
16: return $TS''$
```

4. THE SART IMPLEMENTATION

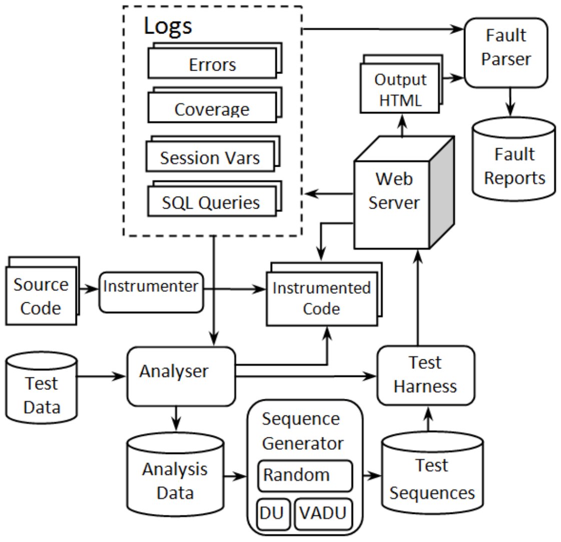


Figure 3: SART architecture

Figure 3 describes the architecture of the prototype tool SART (State Aware Regeneration Tool) with which we implemented our approaches. The Instrumenter instruments the code to record branch coverage. The Analyser consists of a static and dynamic component.

The static component is written in Stratego/xt [7] and PHP-Front [6] and determines session variable and database table Def-Use locations from the source code. The results of the static analysis are stored in the Analysis Data repository. Stratego/xt is a program transformation language and PHP-Front provides libraries for Stratego/xt supporting PHP.

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, executed database queries and statement coverage. 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 [23] to record statement coverage for each test case and then match 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 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 1, 2 and 3 using 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.

5. EVALUATION

We designed our evaluation to answer the following research questions:

RQ1: How many new sequences are generated using VADU compared to DU?

We ask this question to investigate the feasibility of the proposed VADU approach. The number of test sequences generated for the State-based DU approach (DU) depends on the number of definitions and uses present in the code. However, the Value-Aware DU (VADU) approach defined in Section 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 in our study. In answering RQ1, we also examine the results of the static analysis and report the numbers of session variables and database calls.

RQ2: How much can branch coverage be improved?

We ask this question to investigate the effectiveness of DU and VADU in terms of branch coverage. The proposed approaches generate 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. We measure additional branch coverage and compare results for DU and VADU.

RQ3: How much can fault finding ability be improved?

We ask this question to investigate the effectiveness of the approaches; as with other testing approaches, the real test is fault detection, not merely coverage.

For RQ2 and RQ3 we also compare both DU and VADU to random generation of sequences, to provide a baseline for comparison. We also use random sequences as a baseline for comparison because random construction of request sequences has previously been proposed in the literature [12, 26] and so it denotes the current state of the art for web application test suite regeneration.

5.1 Web Applications Studied

We selected 4 PHP web applications that use a database and session variables to perform the evaluation. These applications have been used by other research on web application testing [2, 5]. Table 1 provides a brief description of each application.

Table 1: The web applications used in the study.

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 admin system
Webchess	0.9.0	24	2,701	Online chess game
Timeclock	1.0.3	62	14,980	Employee time tracker

5.2 Experimental Set-up

We use 30 test suites sampled from test cases generated by a tool called SWAT from previous work [2]. All the test suites are chosen to have branch coverage within 10% of the maximum coverage that can be achieved by the tool. We choose test suites with relatively high coverage because increasing coverage for already high-coverage suites is harder than seeking to improve low coverage suites. That is, we wanted to choose test suites to which any increase in coverage or fault detection would be non-trivial to obtain, in order to pose a demanding challenge to our approach.

Table 2: 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 3: Average numbers of generated test sequences (seqs) and requests (reqs) for VADU and DU for both session variables and database tables for 30 test suites for each of the 4 applications.

	Original			Sessions		Database	
App Name	Seqs	Reqs	Alg	Seqs	Reqs	Seqs	Reqs
FAQForge	34	72	DU	2.4	4.9	38.2	76.5
			VADU	7.6	15.1	9.1	145.5
Schoolmate	174	368	DU	6.1	12.1	638.1	1,276.3
			VADU	31.9	63.7	83.4	3,706.0
Webchess	44	95	DU	55.6	111.3	49.6	99.3
			VADU	109.7	219.4	4.4	68.8
Timeclock	244	507	DU	36.0	72.0	0	0
			VADU	133.3	266.5	2.4	102.4
All apps	123	260	DU	25.0	50.1	181.5	363.0
			VADU	70.6	141.2	24.8	1,005.7

We apply the State-based DU (DU) approach described in Algorithm 1 and the Value-Aware DU approach (VADU) defined in Algorithms 2 and 3. We measure and compare the improvements in branch coverage and faults found compared to the original test suite. We also compare the computational cost that was needed to achieve these improvements.

Computational cost is measured as the total number of HTTP requests that we needed to execute to achieve the final improvements in coverage and faults. We believe that the number of requests executed is a better measure of computational cost because it is not affected by the specifics of machine and platform on which experiments are performed. However, we also measure total execution time and report these results 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 we simply call 'Random' hereafter, we generate the same number of test sequences generated for DU and measure coverage and faults found. The implementation for Random insures that only distinct sequences are generated. The way two HTTP requests are combined for Random follows a similar principle to that used for DU depicted in Figure 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.

We use an automated oracle to identify faults revealed by the generated new sequences. That is, our oracle automatically reports PHP and SQL execution errors parsed from PHP error log files and the output HTML pages of each test case. Only faults that are caused by a unique code location and have a distinct type are counted (to avoid double counting of faults). We use an automated oracle to evaluate the approaches because it is unbiased and is unaffected by the experimenters' involvement in the evaluation.

We perform a Wilcoxon paired one-sided signed rank test at the 95% confidence level to determine the statistical significance of the observed results. We use the Wilcoxon test because it is non parametric and we wish 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 3 algorithms starts with the same initial test suite. The test is one-sided because we know that the median of DU is above that of Random and that the median of VADU is above that of DU.

5.3 Results

In this section we present the results obtained from our evaluation on the four web applications for each approach.

5.3.1 *Analysis Result and Number of Generated Sequences*

Table 2 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, we found 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 our 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 3 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.

Table 3 shows the number of sequences generated for each approach. We did not include Random because the number of sequences generated for Random is the same as DU. The table also reports the number of test cases in the original test suite. VADU generates more sequences than DU (as expected) but the average increase in the total number of requests in all sequences is 2.8 times for session sequences (70.6 for VADU compared to 25 for DU) and 2.77 for database sequences (1,005.7 for VADU compared to 363 for DU). As the analysis of computational cost reveals, this increase in test cases is comfortably manageable with reasonable time bounds.

When examining Table 2 we can calculate the number of DU pairs expected. When comparing this number to the generated sequences for DU, we find that in some cases the number of sequences is smaller than expected. We examined this in more detail, revealing two potential causes: Either no test sequence was found to cover some DU pairs or one sequence covers multiple DU pairs. In the case of Timeclock, no database sequences where generated for DU because the one definition and 2 uses (Table 2) belong to different tables, therefore no Def-Use pairs were identified.

One interesting observation is that, for Webchess, the number of sequences and requests generated for database tables using VADU is lower than those generated for DU. 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.

5.3.2 *Coverage*

Branch coverage results are reported in Table 4. 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, we report mean and median coverage (and fault detection) values in the table.

The results indicate that DU performs better than Random for three of the four applications. For Webchess, the mean coverage for Random is better than the mean for DU. What is interesting is that even though the mean coverage improvement for Webchess achieved by Random is 7.75% (compared to 3.19% for DU), the Wilcoxon test reveals that DU significantly outperforms Random. This apparent contradiction between mean and significance tests highlights the importance of understanding the meaning of results from non-parametric statistical testing.

The box plots of the 30 results for coverage improvements for each algorithm are depicted in Figure 4(c). As can be seen, the extreme range of Random is much higher than for DU. However, over all 30 trails in the sample, DU outperforms Random on 21 occasions. This is why the Wilcoxon test indicates that DU is significantly better than Random, a finding reflected in the median coverage improvement values (1.70% for DU, compared to 1.21% for Random).

VADU yields a higher improvement than both DU and Random for all four applications. This is a particularly pronounced effect for Webchess where the mean additional improvement in coverage over DU is 22.12% (25.31% for VADU compared to 3.19% for DU). The difference in median is even higher (33.18% for VADU compared to 1.7% for DU).

When investigating the causes of this strong performance we 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. The DU pairs selected to construct test sequences for DU are not value sensitive, so the chosen pair might not include a definition request for the **gameID** variable that selects a valid

Table 4: 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	2.39	2.11	0.00	0.00	186	36
					DU	**9.01**	**8.85**	**4.12**	**4.00**	233	42
					VADU	**14.10**	**14.21**	4.12	4.00	291	53
Schoolmate	66.32	66.30	96.30	95.50	Random	0.85	0.83	2.00	1.94	2,278	377
					DU	**13.66**	**13.72**	**12.15**	**11.73**	2,615	600
					VADU	**14.42**	**14.34**	**14.31**	**13.66**	4,547	781
Webchess	38.20	38.06	67.83	68.00	Random	7.75	1.21	1.96	0.75	397	170
					DU	**3.19**	**1.70**	**7.22**	**7.35**	465	210
					VADU	**25.31**	**33.18**	**9.30**	**8.82**	537	273
Timeclock	18.11	18.12	177.40	178.00	Random	0.02	0.00	0.00	0.00	559	231
					DU	**1.60**	**1.55**	**1.17**	**1.12**	538	269
					VADU	**5.11**	**5.10**	**9.02**	**8.99**	1,366	615
All apps	47.53	52.60	98.13	74.50	Random	2.75	0.94	0.99	0.00	855	203
					DU	**6.86**	**7.84**	**6.17**	**5.19**	963	280
					VADU	**14.74**	**13.68**	**9.19**	**8.97**	1,685	431

game. These findings confirm the usefulness of our VADU approach.

The top row of box plots of Figure 4 demonstrates the variations in coverage improvement over 30 test suites for each of the four applications for each approach. The difference in performance between the three approaches is clear for FAQForge and Timeclock (Figures 4(a) and 4(d)). In Schoolmate, the improvement of VADU over DU is comparatively lower (Figure 4(b)), though VADU does, nevertheless, perform statistically significantly better.

For Webchess (Figure 4(c)), we notice that the improvement in coverage for VADU can be as small as 1.3% and as high as 43% (over the 30 trails). We examined this peculiarly high variance, finding 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.

5.3.3 *Faults*

Fault detection results for the 30 original test suites and the new test sequences generated by each of the three approaches for the four applications studied are reported in Table 4.

Random only finds new faults in two of the four applications (Webchess and Schoolmate). DU finds an overall mean of 6.17% new faults that were not discovered by the original test suite with a median of 5.19%. VADU performs better than DU in all applications except FAQForge. VADU improves the fault finding ability of the original test suite by a mean of 9.19% over all applications and a median of 8.97%.

Although the improvement in branch coverage for FAQForge is higher for VADU compared to DU, both approaches find the same number of additional faults. We also notice that although Random improves branch coverage, 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.

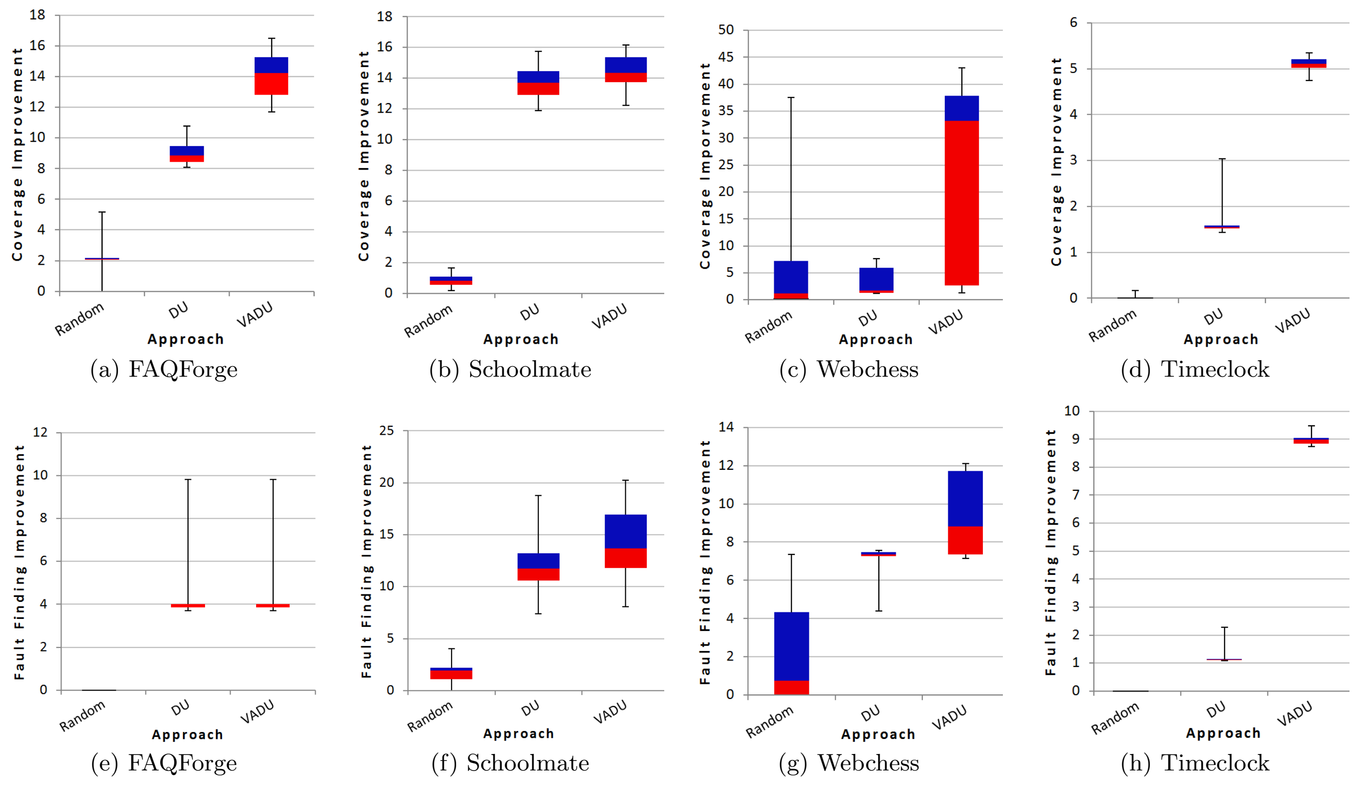
The bottom row of box plots in Figure 4 shows the variations in the percentage improvement in faults found for each of the three approaches over 30 test suites for each of the four applications compared to the original test suites. It is interesting to observe that, for Webchess, although the difference in performance of DU and Random for branch coverage is relatively unclear, in fault finding, it is clearer that DU performs better than Random. This suggests that the way two sets of test sequences are constructed (in this case DU and Random) has an effect on fault detection even when coverage is identical or comparable.

5.3.4 *Computational Cost*

In this section, we report the computational cost of our experiments (executed requests and elapsed time). However, in our 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 finding are presented in Table 4. 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 we needed to execute for the dynamic analysis needed for DU and VADU as well as the execution of the new sequences and the measurement of improvements.

VADU requires the largest number of requests, while Ran-



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

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

Figure 4: Variations in coverage and fault detection improvement results over 30 test suites for each of the 3 approaches on each of the 4 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.

dom requires the least. This result is expected because Random does not need any dynamic analysis and VADU generates the largest number of requests in the new test sequences as examined in Table 3.

Execution time is measured as the total elapsed time for all activities needed to generate and execute the new sequences. Random is the fastest running, on average, in 3.38 minutes (203 seconds) over all applications with the slowest application (Schoolmate) running, on average, in 6 minutes (377 seconds). DU, on average, runs in less than 5 minutes (280 seconds) for all applications. Schoolmate is the slowest executing in 10 minutes (600 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.

5.4 Answers to Research Questions

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

5.4.1 RQ1: How many new sequences are generated using VADU compared to DU?

The empirical evaluation showed that, although VADU generates more sequences than DU, the increase in the number of requests generated for the combination of session variables and database tables is, on average, 2.78 times; this suggests that VADU is feasible. Of course the tester should decide on whether to use DU or VADU based on the time and resources available for testing.

5.4.2 RQ2: How much can coverage be improved?

DU improves mean branch coverage for all four applications studied compared to the original test suite and performs better than Random for three of the four applications. When comparing the median, DU performs better than Random for all four applications. The overall improvement in percentage in branch coverage for DU has a mean of 6.86% over all 30 test suites for all four applications.

For VADU improvement is even higher with a mean of 14.74% and VADU performs better than both Random and DU for all four applications. Statistical testing confirms that these improvements, for DU compared to Random and VADU compared to DU and Random, are statistically significant for all four applications using Wilcoxon paired one-sided signed rank test at the 95% confidence level.

5.4.3 RQ3: How much can fault finding ability be improved?

The empirical evaluation shows that using the automated oracle, DU can increase the number of faults found by a mean of 6.17%, compared to the original test suite (over the 30 test suites evaluated for the four web applications). DU performs better than Random over all test suites and all four applications. VADU increases the number of faults found by a mean of 9.19% and performs better than the other two approaches for all applications. The Wilcoxon paired one-sided signed rank test at the 95% confidence level confirms that these improvements are statistically significant.

5.5 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. We tried to minimize factors that affect the measurement of execution times by sharing code between the implementations of the approaches whenever possible. We also provide the number of requests executed 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. We chose applications that were used by previous research on web application testing and that are also used in current practice. We also selected these applications because they use both session variables and a database. The test suites selected can also affect results. This is why we selected 30 different test suites for each application that have relatively high coverage.

Construct threats: Construct threats are related to the measurements we used to compare the three approaches. We used branch coverage and faults found, 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.

6. RELATED WORK

Dataflow-based approaches to testing have a long history of application to conventional applications, dating back to the seminal work of Rapps and Weyuker [21, 22] and Laski and Korel [18]. Harrold and Rothermel [17] adapted dataflow testing techniques to classes of Object Oriented systems, while, Liu et al. [19, 20] applied a data flow testing approach to web applications.

Liu et al. used traditional data flow modelling of the server code, seeking to generate test cases for structural coverage of web application server-side code. By contrast, our approach 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).

Our argument 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.

Sprenkle et al. [26] 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. [11, 12] also suggested combining parts of different sessions to produce new test suites. These previous studies indicate that recombining or re-ordering test cases can lead to more effective test suites. However, the new test suites were generated purely randomly. By contrast, in this paper we propose an approach to produce these new test suites by analyzing the effect on the server-side state. As our results show, this state-aware approach significantly improves both server-side coverage and fault detection.

Alshahwan and Harman [1] 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 in this paper we consider 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 [1, 2, 4, 5, 14, 15, 24, 28]. 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 our regeneration approach.

Regeneration and augmentation has also been applied to test suites for conventional applications, [3, 25, 29], 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 [8, 9, 10, 13, 16], though these focus on structural coverage of database dependent branches, whereas our approach targets database state interactions for web-based applications.

7. CONCLUSION

In this paper we introduced two approaches to regenerate test sequences from existing pools of HTTP requests present in the original test suites. The approaches we propose exploit server-side state manipulation to generate new test sequences that define and use the state in ways not present in the original test suite. We introduce the Value-Aware DU approach that is aware of the values and specific state modifying SQL statements as well as traditional DU pair information.

We introduced a tool, SART (State Aware Regeneration Tool), that implements our approaches for PHP applications. We report the results of an empirical evaluation on four real world web applications. We report and compare branch coverage information and faults found for the two approaches and also compare to the random recombination approach currently advocated in the literature. Our results provide evidence to support the claim that DU significantly increases coverage and fault detection while our novel value aware approach further improves both coverage and fault detection (at reasonable computational cost).

8. REFERENCES

[1] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the First International Conference on Software Testing, Verification and Validation (ICST '08)*, pages 298–307, 2008.

[2] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering (ASE'11). In *Proceedings of the 26th IEEE/ACM international Conference on Automated software engineering*, pages 3–12, 2011.

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

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 261–272, 2008.

[5] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36:474–494, 2010.

[6] E. Bouwers and M. Bravenboer. PHP-front: Static analysis for PHP. strategoxt.org/PHP/PhpFront.

[7] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[8] M. Y. Chan and S. C. 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.

[9] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An agenda for testing relational database applications: Research articles. *Software Testing, Verification and Reliability*, 14:17–44, 2004.

[10] Y. Deng, P. Frankl, and J. Wang. Testing web database applications. *SIGSOFT Software Engineering Notes*, 29:1–10, 2004.

[11] S. Elbaum, S. Karre, and G. Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 49–59, 2003.

[12] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31:187–202, 2005.

[13] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 151–162, 2007.

[14] W. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 285–296, 2009.

[15] W. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE'07)*, pages 145–154, 2007.

[16] W. G. J. Halfond and A. 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, 2006.

[17] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. *SIGSOFT Software Engineering Notes*, 19:154–163, 1994.

[18] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9:347–354, 1983.

[19] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. Object-based data flow testing of web applications. In *Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, pages 7–16, 2000.

[20] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. Structural testing of web applications. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ICSE'00)*, pages 84–96, 2000.

[21] S. Rapps and E. 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, 1982.

[22] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, 1985.

[23] D. Rethans. Xdebug. xdebug.org.

[24] F. Ricca and P. 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.

[25] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. 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.

[26] S. Sprenkle, E. Gibson, S. Sampath, and L. 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, 2005.

[27] P. Tonella and F. Ricca. Statistical testing of web applications. *Software Maintenance and Evolution: Research and Practice*, 16:103–127, 2004.

[28] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 249–260, 2008.

[29] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 2010.