EFFECTIVE REGRESSION TESTING OF WEB APPLICATIONS THROUGH REUSABILITY OF RESOURCES

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Madhusudana Ravi Eda

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Program:
Software Engineering

November 2018

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

EFFECTIVE REGRESSION TESTING OF WEB APPLICATIONS THROUGH REUSABILITY OF RESOURCES

By

Madhusudana Ravi Eda

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Hyunsook Do
Chair

Dr. Simone Ludwig

Dr. Saeed Salem

Dr. Sudarshan Srinivasan

Approved:

November 14, 2018
Date

Dr. Kendall Nygard
Department Chair

ABSTRACT

Regression testing is one of the most important and costly phases of a software development project. Regression testing is performed to ensure no new faults are introduced due to changes in a software. Web applications undergo frequent changes. With such frequent changes, executing the entire regression test suite is not cost-effective. Hence, there is a need for techniques that can reduce the overall cost of regression testing.

We propose two regression testing techniques that demonstrate the benefits from reusability of existing resources in reducing the costs of regression testing of web applications. Our techniques are based on PHP Analysis and Regression Testing Engine (PARTE) approach that identifies code paths that were modified between two versions of an application. We extend PARTE to introduce two components. Reusable Tests component selects existing tests that can be reused to regression test the modified version of the application, and identifies obsolete tests. Test Repair component repairs such obsolete tests. Our hypothesis is that this combined approach of identifying reusable tests and repairing obsolete tests, can reduce the overall effort of regression testing.

To test our hypothesis, we conducted experiments on real-world web applications. In our first experiment, we learned that there are significant number of input values from the original version of an application that can be reused to test the modified version. Based on this learning, we conducted our second experiment to evaluate if a regression test selection technique can benefit from the reusability of input values. Results from the second experiment demonstrated that reusability of input values minimized the cost of verification of input values in selected tests, and identified obsolete tests. Findings from these two experiments encouraged us to conduct an experiment to evaluate if PARTE approach can be further extended to repair obsolete tests. Results from our third experiment showed that a few obsolete tests can be automatically repaired. Thus, these novel approaches demonstrate the benefits from the reusability of existing resources and shows how further studies can be performed to evaluate approaches that combine one or more regression testing techniques to further reduce the costs of regression testing.

ACKNOWLEDGEMENTS

I am grateful to my advisor Dr. Hyunsook Do for the guidance and continual support. Dr. Do is an incredible mentor who helped me tackle challenges not only in my research but also in my professional and personal life. I am thankful to Dr. Simone Ludwig, Dr. Saeed Salem and Dr. Sudarshan Srinivasan for investing their valuable time in serving on my advisory committee. I express my gratitude to all the faculty and staff at the Computer Science department and the University for offering the high-quality education and preparing me for a successful career. I would like to thank my wife, Vani and my daughter, Teju for their support and encouragement throughout this journey. I am thankful to all my colleagues at Microsoft for their curiosity and discussions that greatly motivated me to pursue this research. Special thanks to Dan Leeaphon, my former manager, who instilled the confidence in me to embark on this journey. Finally, I am grateful to my current manager, Vika Teunissen, for accommodating my schedule while preparing this dissertation.

DEDICATION

To my parents *Eda Madhusudana Rao* and *Eda Vasanthi*, whose belief in the transformational power a high-quality education has not only on the individual but also on the well-being of the community, enabled me to pursue a doctoral degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

One of the most important and costly phases of the software engineering lifecycle is the testing phase. Testing ensures that the software matches its specification. Testing presents various challenges, which is the reason why testing remains one of the most widely studied and practiced areas [68]. One of the challenges is because software evolves. Regression testing is performed to ensure no new faults are introduced due to changes in a software [65]. Regression testing of web applications presents a unique challenge that arises due to their development lifecycle [80]. Web applications, specifically commercial ones, undergo rapid and frequent changes. These changes are due to various factors such as the availability of new technology, responding to user feedback, and fixing security issues [31]. Broken features or security defects that are not fixed quickly can harm an organization's reputation, and customers would lose trust in the organization and that eventually can lead to financial loss [10].

The inherent characteristics of a web application makes its maintenance more challenging than for a traditional desktop application. Web applications are dynamic, meaning that the content of a web page changes based on the user or the time accessed. Web applications are designed to operate on multiple platforms. Web applications are heterogeneous, meaning that several technologies are used in each application. Additionally, the ubiquitous presence of web applications makes regression testing of these applications incredibly challenging [99, 48]. Thus, software practitioners and researchers who maintain such applications need appropriate tools that can analyze and effectively test changes in a timely manner.

To date, researchers have presented many techniques and tools to tackle the challenges of regression testing of web applications [52]. For example, Ricca and Tonella [79] proposed *ReWeb* and *TestWeb*, which perform static analysis on a UML representation of a web application and generate test cases covering the modified code paths. Dallmeier et al. introduced *WebMate* [22], which is a tool to detect changes between two versions of an application and to identify any cross-browser incompatibilities. *WebMate* creates a usage model, which is a graph where nodes are the states of application and edges represent the interaction that causes a change of state, and this usage model enables users to systematically explore modifications between two versions of an application and to prepare regression tests that execute the modified interactions. Taneja et al. [90] presented *eXpress*, which uses a path-exploration-based test generation technique to identify the differences between two versions of an application, and prune irrelevant paths from exploration. Thereby,

the number of paths to generate tests is minimized. However, such dynamic symbolic execution techniques are expensive and do not scale.

Do et al. [25] introduced the PHP Analysis and Regression Testing Engine (PARTE). PARTE follows a strategy of generating test cases that cover only the modified code paths. It begins by determining the blocks of code that were modified, then uses program slices and constraint resolution techniques to generate tests cases. Through experiments on real-world applications, we found that PARTE approach is cost-effective, meaning the number of test cases required to cover the modified code paths is substantially reduced. The experiments also showed that the cost of solving constraints, to determine the input values for generated tests, is high and it often requires manual intervention even when automated solvers are used. For instance, solving an input that contains a long string can take an inordinate amount of time [45, 39]. We learned that reusing input values can reduce the cost associated with constraint solving. A technique to reuse constraint values was demonstrated in the Green solver interface [93].

A limitation in this area of research is the lack of empirical evaluation of approaches that explore the benefits of reusability of existing resources in regression testing of web applications. Regression test selection, prioritization and minimization techniques enable software practitioners to reuse existing test suite [102]. Empirical evaluation of how these techniques can be augmented with constraint reuse information to further reduce the regressions testing costs, is not available in the published literature. For instance, our recent research showed that many constraint values from the original version of a web application can be reused to regression test the modified version of that application. Thereby, the cost of solving for constraints is minimized [40]. Regression test selection technique could be augmented with constraint reuse information to identify which of the selected tests have reusable inputs. Thus, the cost of test input value verification either manually or through actual test execution can be minimized. Tests that do not have reusable input values will be obsolete, and more savings can accrue if such obsolete tests can be automatically repaired and reused. Hence there is a need to develop techniques that explore the benefits of reusability of existing resources, and empirical evaluation of such techniques will provide guidance for researchers and software practitioners in reducing the cost of regression testing of web applications.

1.1. Goal of this Dissertation

Our thesis is that reusability of existing resources during regression testing can result in significant savings that are accrued over multiple releases of a web application.

Failure to ignore reusable resources will result in costs towards constraint resolution, verification of test inputs in selected regression tests and determining the cause of failure in a test. Performing these tasks manually can result in additional costs and errors. Hence, if our thesis were true, then researchers and software practitioners can reduce the costs and manual errors by reusing existing resources to test web applications effectively, and perform test maintenance. Though there will be upfront costs in setting up the infrastructure that can enable such reuse, the costs can be offset due to the savings that accrue over frequent releases of the web application.

1.2. Approach to Achieve the Goal

To address the stated goal, we present our plan of approach to conduct the empirical evaluation to demonstrate the benefits of reusability of existing resources in reducing the overall costs of regression testing of web applications.

We use the terms 'cost' and 'effort' interchangeably to refer to the resources needed for conducting regression testing. These terms refer to the resources needed in the absence of automated tools to perform test selection, identify reusable tests, identify obsolete tests and perform test repair. We use the term 'reusable constraint value', which satisfies the constraints in a regression test path, interchangeably with the term 'input value' when there exists an input value for a test that covers the regression path.

First, we propose a technique that identifies reusable constraint values for the modified version of the program. Second, we propose a regression test selection approach that selects a subset of existing tests that are necessary to test the modified version. Some selected tests cannot be used for the new version if their inputs are not applicable anymore, but finding such inputs is time-consuming. Thus, we apply the technique that identifies reusable input values for the selected tests so that we can identify tests with reusable inputs. If a small portion of the new version is modified, then we can expect greater benefits associated with this technique. Third, we propose a test repair technique that attempts to repair tests that are not applicable to the new version (obsolete tests). A minor change in the product can cause up to 74% of existing tests to fail [61]. Some of the failing tests can be fixed with minor updates (e.g., parameter data type or order change) in the test code, but performing such maintenance tasks can cost an organization up to $120 million a year [33]. Automating a test repair task will allow developers to minimize the manual inspection and correction of failing tests, and reduces the cost of maintenance of the test suite.

The proposed constraint reuse technique compares the definitions and uses (def-use) of a variable in the original and modified versions of the program. If def-use chain of the variable is same between two

versions, then the constraint value for that variable can be reused. The proposed test selection technique uses slice information obtained through PARTE to identify code blocks that were modified. Existing test coverage information enables test selection to identify tests that cover the modified code paths. The selected tests could contain inputs values that are not applicable to the modified version. To identify such input values automatically, each input variable in the selected test set is compared with the set of variables that have the same def-use chain in the original and modified versions of the program. Tests that have reusable constraints are included in the set of regression tests to be run against the modified version of the program. Tests with non-reusable constraints are identified as obsolete tests. The proposed test repair approach automates two of the most common test case maintenance activities. Those two activities are fixing assertions [23] and updating method invocations [63]. Program dependence graphs (PDGs) of source and test code are analyzed to identify assertions and method signatures. Based on the obsolete test's failure message obtained from PHPUnit [73], an appropriate repair is identified, and the repaired copy of the test is created.

To assess the proposed approaches, we performed three controlled experiments. In the first experiment, we investigated whether reusing constraint values can reduce the overall cost of regression testing. We conducted the experiment on five open-source PHP web applications. Each application had multiple versions. The results from this experiment show that a large portion of constraint values can be reused for most versions in all five applications. Thus, the cost of solving constraints could be substantially reduced. In the second experiment, we investigated the cost-effectiveness of regression test selection with constraint value reuse using four open-source PHP web applications with multiple versions and unit tests. The results from second experiment show that large savings can be achieved by selecting a small portion of existing test cases, but the results varied across programs and versions. We also observed that the rate of constraint value reuse is proportional to rate of reusable tests across most versions of each application. As constraint values remain applicable for multiple versions, our approach offers greater cost savings. In the third experiment, we assessed if the proposed test repair approach is effective in repairing the obsolete tests that were detected through the test selection experiment. From the results of the experiment, we observed that the percentage of obsolete tests that were correctly repaired is high, which means the cost of manual effort in repairing tests can be reduced. We also learned that failures due to changes in method parameters is not common while minor changes in the product can affect several assertions in the test suite.

1.3. Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 discusses the background and related work. Chapter 3 describes the overall methodology of our research. Chapter 4 describes the infrastructure developed and our approach to conduct the controlled experiments. Chapter 5 presents the details of the three experiments we conducted. Insights and practical implications from the results of the three experiments is summarized in Chapter 6. Finally, Chapter 7 presents the impact, and the future directions of our research.

2. BACKGROUND AND RELATED WORK

This chapter presents the background and related work of our research. We briefly describe our motivation, describe the regression testing concepts applicable to work described in this dissertation, and how our research relates to the existing research.

2.1. Motivation

I was greatly motivated with my experience over the past 10 years working as a software engineer at Microsoft. I was a member of the team that develops the product called *Microsoft Visual Studio*, which is an integrated development environment (IDE) [94]. Currently, I am on the team that develops *Microsoft Dynamics AX*, which is an Enterprise Resource Planning (ERP) product [8]. Though these two products cater to different set of customers, one of the most critical and resource consuming phases in the development of these products is the regression testing of all the changes before the product can be delivered to the customers.

A decade ago, the time between two major releases of *Microsoft Visual Studio* was around 3 years. This gave sufficient time for the engineering teams to regression test the changes by running the entire test suite. To meet the current customer needs and adapt to the rapidly evolving technology, a minor release of the latest version of *Microsoft Visual Studio* happens every six weeks, and certain bug fixes can be delivered weekly [77]. To enable such faster releases, I added attributes to tests so that tests can be filtered based on those attributes. These attributes captured data such as the features and the importance of those features covered by the test. A detailed history of test execution including traceability of a test failure and the corresponding bug reports, enabled faster analysis of test failures. Thus, I was able to reduce the turnaround time and minimize the cost of regression testing of the features that I owned in *Microsoft Visual Studio*. Anderson [3] described how *Microsoft Dynamics AX* used multiple data sources and techniques to determine the effective subset of existing tests that are needed to regression test each code submission.

As described in Chapter 1, web applications pose additional challenges compared to their desktop counterparts. This sparked my interest in researching cost-effective regression test approaches for web applications.

2.2. PHP

Hypertext Preprocessor (PHP) continues to be one of the most popular programming languages [76]. Around 80% of the existing websites are built using PHP [95]. Dynamic typing, flexibility to have HTML and PHP script in a same file, and an ecosystem that provides a large set of predefined libraries to simplify development are some of the reasons for the popularity of PHP web applications.

PHP is a weakly typed language, meaning the arguments passed to a function does not have to match exactly with the expected type in the function's signature. Weak typing gives the developer some flexibility, for example, minimizes the number of overloads of a function since one version of the overload will be able to accept and return different types. This flexibility allows reuse and improves the speed of development. On the other hand, this flexibility introduces the risk of a variable changing its type during the execution, which leads security bugs [21, 75]. To ensure static typing to PHP variable such that a variable maintains a single type throughout execution, *Hack*, a new variant of PHP was developed at Facebook [29]. Though this reduces the possibility of bugs during the development, modifications to the data type or the order of the parameters of a function can cause test failures.

Given the popularity of PHP and access to source code of PHP based applications, we chose to pursue our research in finding cost-effective regression test techniques for web applications developed using PHP.

2.3. Impact Analysis

Impact analysis is the process of identifying the effects of a change in a software system [6]. Impact analysis is used not only in testing but also in tasks such as traceability of requirements, cost estimation and resource planning.

Tonella [92] proposed an approach that combines the decomposition slice graph and concept lattice to determine the impact a variable may have on other variables in a program. A decomposition slice graph shows the dependency between partitions where a partition is a computation performed on different variables in the program. A concept lattice, groups entities in a program, such as methods, based on common attributes. These groups are organized into a hierarchy of concepts that are related either through a generalization or a specialization. In the real-world where a change request is usually written in natural language (e.g., English), impact analysis can be performed to determine the impact set of the change request. Ren et al. [78] presented a tool that performs structural static analysis [89] on Java programs to identify the regres-

sion tests. Ziftci et al. [103] applied impact analysis to detect the code change that introduces test failures in the Continuous Integration (CI) system at Google. Bell et al. [9] presented a technique called *DeFlaker* that uses static impact analysis and other attributes to mark a failing test as flaky if the test does not cover the path impacted by the code change.

Gethers et al. [32] proposed an integrated approach to perform impact analysis given the textual change request. Their approach begins with mining the related change requests to determine the software entities (e.g., classes, methods, etc.) that should be included in the impact set. Second, use an information retrieval technique to mine software artifacts such as bug reports to identify software entities that should be added to the impact set. Third, use dynamic analysis such as execution traces to estimate the impact set for the given change request. Kabeer et al. [43] conducted an industrial case study where impact analysis was used to estimate the effort needed to perform a change request. They found that effort and duration can be predicted with an accuracy of 84% and 72%, respectively when impact analysis was applied over textual change requests.

Impact analysis should be cost-effective, meaning the cost to perform the analysis should be less than the savings obtained from using the results of the analysis. There are two types of impact analysis - Static and Dynamic. Static impact analysis assumes that a program can take any of the possible paths. Though this is safer, this approach produces a large impact set. In contrast, dynamic impact analysis that uses runtime information of the program, can produce a more relevant impact set. This conciseness makes dynamic impact analysis more preferable. However, dynamic impact analysis techniques are either fast but imprecise or precise but slow. The challenge of balancing this trade-off has attracted lot of researchers. Cai et al. [11] introduced *DiaPro*, which is a framework that postulates that a hybrid approach of unifying static and dynamic impact analysis can be cost-effective. In *DiaPro*, the process of impact analysis begins by performing a static impact analysis to generate the program dependence graph and instrumented version of the source code. Next, a dynamic impact analysis is performed to capture execution traces. Finally, a unified phase where the impact set determined through static analysis is pruned based on the trace information obtained during dynamic impact analysis.

In our work, we used static impact analysis to not only identify the regression tests but also identify obsolete tests and repair those obsolete tests.

2.4. Regression Test Selection

Regression test selection (RTS) techniques reduce testing costs by selecting a subset of test cases from an existing test suite (surveyed in [44, 28, 81, 102]).

Early work conducted by Lenung et al. [51] and Rothermel et al. [82] established a theoretical foundation for RTS techniques and a framework for empirical evaluations of these techniques. Initially, RTS techniques focused on testing procedural languages using source code, but in recent years, the focus of RTS shifted to various complex, real-world application domains. For example, Kim et al. [46] presented an RTS technique aimed at ontology-driven database systems because traditional RTS approaches cannot be applied for such systems. The proposed technique builds graphs for the original and modified ontologies, compares the two graphs to identify the changes, and selects test cases using the changed information. Mirarab [62] introduced a size-constrained regression test selection technique. The proposed technique selects a subset of test cases with the predetermined size of tests using multi-criteria optimization. This approach can use limited resources more effectively when time constraints on regression testing are imposed.

Some RTS studies employed hybrid approaches to further improve regression testing processes. For instance, Shi et al. [87] performed an extensive study of combining RTS with test-suite reduction. The proposed approach improves the turnaround time of regression testing, and requires fewer resources than running the full test-suite. Another study by Elbaum et al. [27] proposed an approach that combines RTS with Test Case Prioritization (TCP) in the CI system at Google. Traditional RTS and TCP techniques that rely on code instrumentation and discrete sets of tests cases, were modified to operate on data available for a given time window to accommodate the frequent testing requests in CI.

More recent RTS studies demonstrated the effectiveness of RTS approaches considering both benefits and associated costs. The improvements in effectiveness obtained by certain techniques do not guarantee the practical cost-effectiveness of those techniques because the techniques also have associated costs. Legunsen et al. [50] conducted a study to evaluate how RTS techniques perform in terms of the number of tests selected, time overhead, safety, and precision. Őqvist et al. [67] proposed a dependency graph extraction-based RTS technique that attempts to reduce the overhead of running RTS in an agile development practice where regression testing is performed frequently, and the code changes occur in a limited set of files. Herzig et al. [38] introduced *THEO*, a cost-based test selection strategy that skips execution of test if the cost of

execution is greater than the expected cost of not running the test. *THEO* ensures all tests will be executed before the software is ready for release.

In our research we use perform RTS based on the regression paths identified through PARTE. We augment RTS with reusable resource information to filter out tests whose input values cannot be reused in the modified version of the application under test.

2.5. Test Repair

Test repair is the process of repairing a failing test.

As software evolves, an existing test can fail either due to a bug in the source code or the test becoming obsolete [36]. If the failure is due to the obsoleteness, then repairing the test and putting it back to use would retain the original intent of the test. Throughout the lifetime of a real-world application, test suites are modified so that the tests are effective in identifying regressions while the application evolves. Manually modifying a test is tedious and error prone process. In practice, obsolete tests can account for 26% of test suite [49]. This motivates research on obsolete test repair techniques that can assist a developer in devising fixes for source code [66, 60, 70] or tests [23, 91, 63, 35].

There are several tools available for repairing web application tests. For example, *Web Application Test Repair (WATER)* [14], automatically identifies the mismatch between the expected and the actual values of a web page elements, and identifies the elements that are displaced or modified. This information enables the developer to make appropriate changes to tests. While *WATER* performs repairs by comparing two version pairs, a more fine-grained approach that considers the incremental changes than happen between the versions was presented in *WATERFALL* [35].

Pinto et al. [74] observed that test repairs that involved only fixing an assertion, was less than 10% of the total repairs in the open-source applications they studied. In contrast, it is more likely to find a test repair that requires analysis of changes to method calls, such as method parameter added, deleted and modified, and synthesize new calls so that the existing assertions in the test can be successful. Daniel et al. [23] developed *ReAssert*, a JUnit test repair tool that uses various strategies including symbolic execution, to determine the actual value in the assert statement in the obsolete test so that the assertion is successful. Mirzaaghaei [63] presented *TestCareAssistant (TCA)* that analyzes method signatures to detect the changes in parameters and determines an appropriate fix for the obsolete test.

In our research, we repair obsolete tests not only by fixing assertions but also repairing method signatures. Static impact analysis information allows us to identify the offending assertion or the parameter in a method signature.

2.6. Program Repair

Program repair techniques focus on modification to the source code of the software to address a defect. A defect is the difference between the expected behavior and the observed behavior when the software is executed. An *oracle* provides the expected behavior of the software [88]. Unlike behavior repair, a state repair is a fix performed while the software is being executed. When a software is detected to be in a faulty state, then actions such as reinitializing and rebooting the whole or a part of a system and rollback to a last known good state, can get the system out of the faulty state.

In published literature, there are many program repair techniques proposed and tools implemented [64]. *GenProg* [30] is a genetic programing based program repair tool. This tool uses mutations such as insert, delete and replace, and crossover operators to identify the variant of the program under repair that allows the failing (negative) test to pass and does not introduce new failures in the passing (positive) tests. Chandra et al. [12] presented a tool called *ANGILINA* that implements angelic debugging, which is an approach to locate and fix faulty expressions in the program under test. Angelic debugging begins with the programmer who based on his/her experience, fault localization tools or intuition would select the scope of the source code where a modification could fix the defect. Within this scope, the tool would try to identify plausible expressions that can be repaired. For each expression, the tool finds an alternate value that can cause the failing test to pass, and the value should not cause a previously passing test to fail, meaning no new regression is introduced. If no such value is found, then the developer should try to find the alternate value or select a different or wider scope of the source code. Nguyen et al. [66] presented *SemFix* a program repair tool based on semantic analysis of the program under test. *SemFix* uses three techniques. First, uses a statistical fault isolation tool to determine the statement in the program that needs to be fixed. Second, infer the expected behavior of the identified faulty statement. Finally, update the faulty statement based on the inferred behavior. Mechtaev et al. [60] introduced a tool called *DirectFix* that focuses on generating simple fixes. *DirectFix* uses an approach that integrates fault localization and program repair so that the problem is reduced to solving a partial maximum satisfiability problem. A fix is synthesized such that the repair is simple, meaning the changes to the structure of the program is minimal.

A limited number of program repair tools for PHP web applications are described in the literature. For example, Samimi et al. [85] introduced *PHPQuickFix* and *PHPRepair*, which are tools to repair HTML generation errors in PHP web applications. *PHPQuickFix* identifies errors such as mismatched HTML open and close tags and special characters that are not escaped. Given the expected HTML output for a PHP program, *PHPRepair* can identify and repair the program, using string constraint solvers, to ensure that the PHP program generates the expected HTML output.

Though program repair is not used in our research, the ideas in this area motivated us to think about obsolete test repair strategies.

2.7. Test Suite Augmentation

Test suite augmentation is the process of creating test cases to cover the modified code paths identified during *Impact Analysis*.

The process of identifying modified code paths, assessing whether existing tests exercise affected paths, and preparing new tests to cover those paths has attracted lot of research [5, 86, 101, 100, 90, 13]. Apiwattanapong et al. [5] and Santelices et al. [86] presented a propagation-based approach that uses program dependence analysis and symbolic execution to compute test requirements to assess and augment existing test suite. Xu et al. [101, 100] presented directed test suite augmentation techniques to create tests that cover modified code by utilizing tests selected through RTS to seed a concolic test generation approach or a genetic algorithm to generate new tests. Taneja et al. [90] proposed an approach called *eXpress* that generates tests based on dynamic symbolic execution. To optimize search strategy, paths that are unlikely to detect behavioral differences are pruned, meaning not explored. Thus, this approach focuses on regression test case generation making it very efficient. A model-based regression test suite generation using dependence analysis was proposed by Chen et al. [13]. Information obtained from existing unit tests was used to automatically augment test suites in the framework proposed by Rubinov et al. [84].

The focus of the approaches described thus far was on desktop applications developed using languages such as C or Java. The focus of our research is the regression testing of web applications, which presents a different set of challenges mostly due to the inherent nature of these applications, with characteristics such as multi-tier architecture, multiple technologies, dynamic programming languages and a larger surface area susceptible to attack requiring shorter turnaround time for security fixes. These challenges have attracted many researchers, and several test case augmentation techniques have been proposed. A concolic approach to generating test cases for PHP applications was proposed by Artzi et al. [7] and Wassermann et

al. [96]. Use of crawlers and spiders to identify changes between versions of web applications and generate tests to exercise the modified behavior has been a popular approach to test the interfaces of web applications. For instance, Ricca et al. [79] proposed using a crawler to discover the link structure of a web application. This structure is utilized to produce test specifications for the new version of the web application. Deng et al. [24] proposed an approach of extending *AGENDA*, a set of tools for testing database systems, to test databases of web applications. A web application is represented as a graph on which nodes represent URLs and edges represent links between URLs. Tests are generated for selected paths in this graph. Halfond et al. [34] proposed an approach that uses symbolic execution to identify precise interfaces between web components. Alshahwan et al. [2] introduced two def-use approaches for generation of new tests based on existing tests for web applications. Defining the state of the web application and the corresponding use of that state forms the def-use pair.

Our review of the published literature showed that no attempt was ever made to augment RTS with reusable constraints and test repair. Thus, our research is unique since it combines a constraint value reuse technique with RTS to identify reusable and obsolete tests, and repairs the obsolete tests in a PHP based web application.

3. METHODOLOGY

To support the proposed approach, we extended PARTE [25] to include two new components. First, we implemented a *Reusable Tests* component that consists of *Constraint Reuse* and *Test Selection*. *Constraint Reuse* as described in [25] compares two sets of test paths (original and modified versions) of the system under test (SUT) to identify reusable constraint values. *Test Selection* selects existing tests that cover the modified code paths using the *DejaVu* [83] algorithm, which is a static and safe test selection algorithm. Second, we implemented a *Test Repair* component (PHP Obsolete Test Repair (POTR)) that consists of *Method Signature Analyzer* and *Assertion Analyzer*. *Method Signature Analyzer* (MSA) compares the method parameters in the original and modified versions of the SUT to detect changes in the parameters that caused a test to become obsolete. *Assertion Analyzer* (AA) updates the test oracle by parsing the PHPUnit assertion failure messages to identify the value returned by the SUT. Based on the analysis results, POTR presents a repaired version of the obsolete test.

Before we describe these components in detail, we provide a brief overview of PARTE. [1]

3.1. Overview of PARTE

In Figure 3.1, the boxes depict the main activities. The ovals are external tools that PARTE utilizes. The external tools are PHC and PHPUnit. The dark gray boxes are the new additions to PARTE. These additions are *Reusable Tests* and *Test Repair*. In this section, we describe *Preprocessing* and *Impact Analysis*. In the next sections, we will explain the *Reusable Tests* and *Test Repair* components.

The *Preprocessing* converts the original and modified versions of the SUT into an Abstract Syntax Tree (AST). *Preprocessing* consists of two components: *File Preparation* and *AST2HIR*.

3.1.1. File Preparation

PARTE uses ASTs to generate PDGs. To create ASTs from PHP programs, we used a PHC (PHP compiler), a publicly available, open-source tool. PHC was used because it generates ASTs that contain detailed information about variable data type that is necessary for input data constraint gathering and resolution. However, there are a couple problems with PHC. PHC cannot properly parse '*include*' and '*require*' functions that contain variables instead of constant strings. Further, PHC cannot evaluate the expressions

[1] While PARTE was implemented for PHP web applications, its approach could be applied to any system for which the required information is available.

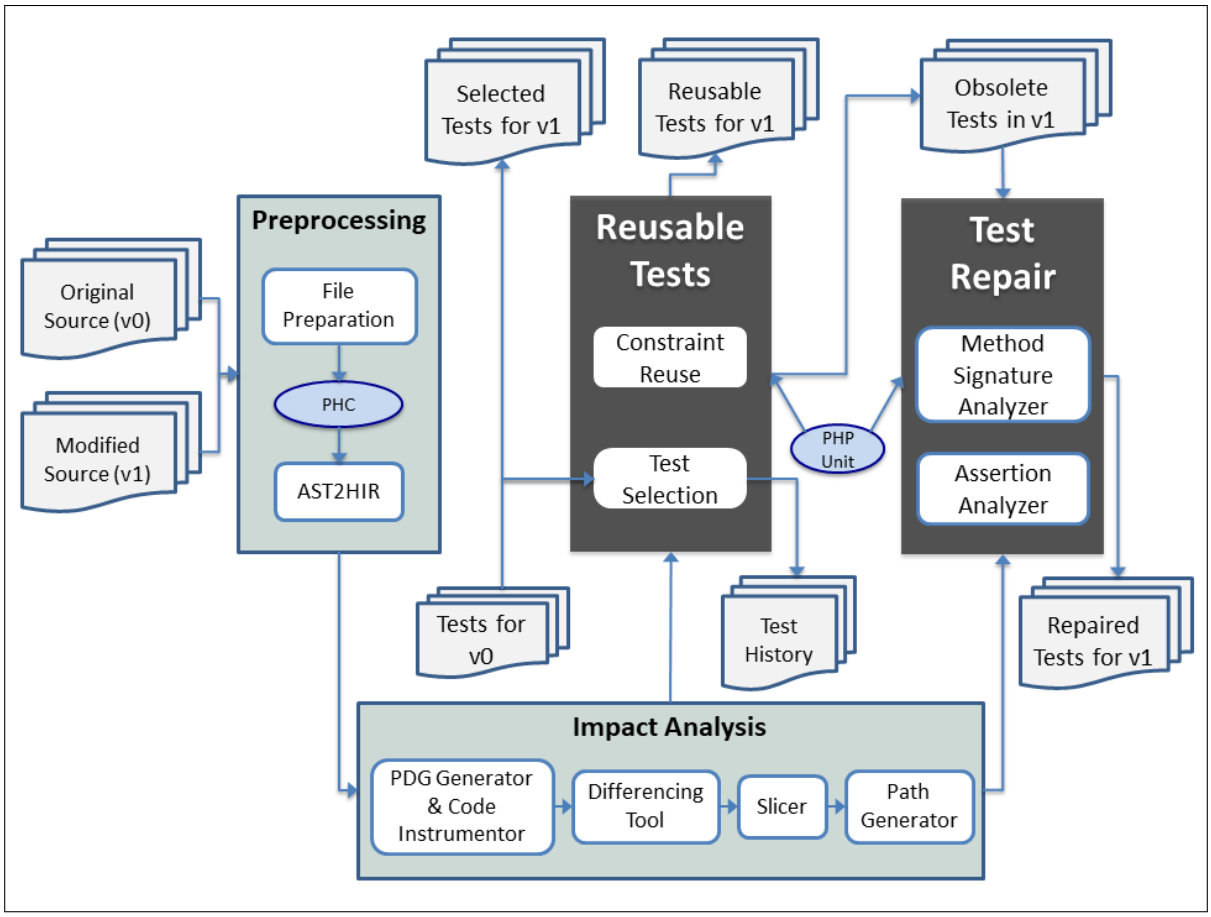

Figure 3.1. Overview of PARTE.

if any concatenation or defined values are used in the '*include*' and '*require*' function. Using defined values and runtime concatenation to include a specified file at a location defined during the web application's installation is a common practice.

PHC cannot resolve non-trivial '*include*' and '*require* function calls. Hence, we built a parsing tool to search for those function calls in the web application's source code and then to locate, define, and store them in a dictionary. When the parsing tool locates '*include*' and '*require* function names, it resolves them using the dictionary that maintains defined values. If '*include*' and '*require* functions are completely resolved, then the tool inserts the corresponding code into the source code file. This process is repeated until the tool visits every '*include*' and '*require* function.

Another common feature of many dynamic web applications is to specify which files to include at runtime. One typical example of this feature is that a user selects a language preference when the web application is presented. For cases like this one, a constant value is manually assigned to the environmental variable in the application.

3.1.2. AST2HIR

After obtaining the preprocessed PHP files from the previous step, ASTs for these PHP files were generated using PHC. Next, AST was converted to a High-level Intermediate Representation (HIR) to simplify PDG generation.

PHC already provides a function that generates HIR code from an AST, but the HIR code generated by PHC does not preserve consistent variable names across versions of the same application, so it is not suitable for the proposed regression testing approach. PHC's HIR is focused on code optimization for performance efficiency. PHC uses a typical three-address code generation approach and creates variable names by incrementing a numerical suffix on a fixed prefix. Thus, if a single statement in a program is added, deleted, or modified, every statement that comes after it in the internal representation could have variable names that do not correspond with the variable names in the original version's internal representation. The proposed regression testing approach requires consistent variable names to properly generate an impact set given two different versions of the PHP program. To resolve this issue, *AST2HIR* tool was implemented to maintain consistent variable names while generating a three-address code representation for the program. Instead of simply incrementing based on the order variables that appeared in a source file, a hashing function was used on the values from the control dependency information combined with the expression values within an AST statement to determine variable names.

There are two main symbol types in the AST: statements and expressions. Statements are the symbols that define classes, interfaces and methods as well as the control flow of the program. Expressions are the rest of the symbols, some of which include arithmetic operators, conditional expressions, assignments, method invocations, and logical operators. The AST that PHC generates is in XML format, and the *AST2HIR* converter parses the XML tags in a depth first manner. When a complex statement ('complex' being defined as an expression that is not in three-address code) is encountered while parsing, it is split into multiple expressions such that all parts are not complex. These parts are given a unique variable name, and assignment expressions are generated to assign the value of the non-complex parts to their corresponding unique variables. Some statements are also transformed to work in the *PDG Generator*. For instance, a 'for' loop is simplified into a 'while' loop. 'Switch' statements are converted to equivalent code using a 'while' loop, 'if' and 'break' statements.

The *AST2HIR* tool then produces a PHP program that is functionally equivalent to the original program but in the high-level internal representation. An AST for this HIR program is then created using PHC. This HIR version of the AST is used to generate the PDGs. Throughout the dissertation, this HIR version of the AST is simply referred to as AST.

3.1.3. Impact Analysis

The *Impact Analysis* component consumes the ASTs from *Preprocessing* and generates Program Dependence Graph (PDG), instrumented code, differences, program slices, and regression test paths. Paths, PDGs and slices produced from *Impact Analysis* are used in *Reusable Tests* and *Test Repair*. *Impact Analysis* consists of the following four components.

1. *PDG Generator & Code Instrumentor*: This component reads the ASTs to generate control flow graphs for all method and function calls.

2. *Differencing Tool*: This component identifies the PHP files that were modified between the original and modified versions of the SUT. For each such modified file, it compares the corresponding PDGs from the original and modified versions to collect all the differences, which includes additions, deletions and modifications.

3. *Slicer*: This component consumes the modification information from *Differencing Tool* to calculate the code change impact set by performing forward and backward slicing to produce a combined program slice for each modified PHP source file.

4. *Path Generator*: This component consumes slices to create linearly independent test paths, which is a set of paths through the application where each path has at least one new edge that was not included in any other paths in the set.

All the components were implemented using Java and we prepared Perl scripts to automate the interaction between the components.

To analyze the impact of software modifications on a web application, we implemented a static program slicer. The benefit of using a program slicer is that it provides a complete way to reveal which areas of the application will be affected by software modifications [1].

Using the ASTs constructed during the preprocessing steps, impact analysis is performed to identify areas of the application that have been affected by code changes. For impact analysis, three steps are involved: PDG generation, program differencing and program slicing.

To build PDGs from ASTs, the approach by Harrold et al. [37] is utilized. First, the *PDG Generator* constructs control flow graphs. During this phase, the *PDG Generator* builds control flow graphs for all methods and functions that have been declared. To construct an inter-procedural control flow graph, all method or function calls within the control flow graphs are linked to their corresponding control flow graphs. Next, the *PDG Generator* analyzes the data by performing a def-use analysis and gathers data dependency information for variables.

After constructing PDGs for the two consecutive versions, it is needed to determine which files have been changed between the versions and which statements within those files have been changed. To perform this task, a textual differencing approach similar to that of the standard Unix '*diff*' command is used. With this approach, the differencing tool compares two versions of PDGs, in XML format, which contains the program structure and statement content. The differencing tool analyzes each program and applies the longest common subsequence approach on the statements' PDG representation. After the tool determines which statements have been edited, added, or deleted, it provides this information to the program slicer.

Finally, the program slicer creates program slices by using the program's modification information. Program slicing is a technique that can be used to trace control and data dependencies within an application [97], and in our work, it is used to calculate the code change impact set. This program slicer performs forward slicing by finding all statements that are data dependent on the variables defined in the modified statement and then produces a program slice following these dependencies. It then performs backward slicing on the modified statement to determine the statements upon which the modified statement is data dependent. For each modified statement, the forward and backward slices are combined into a single slice.

3.1.4. Constraint Collector

Impact Analysis generates test paths that only contain input parameters without the actual input values. The input values need to be determined and assigned to the parameters to make test paths executable. *Constraint Collector* collects constraints for the input values needed to execute the path. Inputs for *Constraint Collector* are the PDGs and the paths obtained from the *Path Generator*. The top-level activities in

the constraint collection are parsing the path and PDG, which are XML files, identifying constraints for each path, and writing the collected constraints to an XML file.

Constraint collection begins with path nodes that contain a conditional statement. The primary activities to collect constraints are determining the truth value of conditional statements and reducing the conditional expression in these statements. A path constraint corresponds to a path PDG node with an 'if' statement. To determine the truth value of a constraint, the collector looks ahead one node in the path node list and examines the node type (which will be either true or false). After determining the constraint truth value, the constraint condition is then recursively reduced. This reduction involves parsing the expression string to determine the expression type (using regular expression matching), creating an expression object with this type, assigning any expression attributes from the parsed information (e.g., operator type) to the expression object, and generating appropriate expression objects for child expressions, if any. If there are child expressions, they are recursively reduced in the same way.

Each type of expression provides its own method for reducing child expressions in order to take advantage of information on the reduction context. For example, a compound boolean expression must have child expressions that are boolean. When generating child expressions, this information can be provided in addition to what is provided by regular expression matching. This is useful in verifying that generated expressions are of the correct type. If regular expression matching determines that an expression is a variable (e.g., '*$num_of_files*'), then the reduction process involves additional steps:

1. The collector backtracks in the path node list, starting at the variable node's index. It continues backtracking until a PDG node that provides a definition of the variable is encountered or until the beginning of the list is reached. If a definition is found, then the expression generated for the variable is of the complex type. This type is used for variables that can be defined in terms of other expressions. If no variable definition is found before reaching the start of the list, the generated expression is of the simple variable type.

2. Regular expression matching is performed on the variable expression string to determine if it is indexed (e.g., '*$files[$id]*'). These correspond to PHP array variables. An expression is then generated for the array's index, and the generated expression for the variable is an indexed version of the type (i.e., simple or complex) determined via backtracking.

The recursive reduction process terminates on expressions of the simple type because simple type expressions do not contain child expressions that need to be reduced. After collecting a constraint, the collector records it for inclusion in the output. Once all path constraints have been generated, the collector writes the constraint information to an XML file.

3.2. Reusable Tests

As shown in Figure 3.1, *Reusable Tests* consist of two components: *Constraint Reuse* and *Test Selection*. *Constraint Reuse* determines a list of variable values that can be reused in regression test cases [40]. *Test Selection* performs regression test selection to determine a subset of tests that are applicable to the new version of the program, including identifying reusable input values for the selected tests. We describe these two components in detail, including algorithms and an example.

3.2.1. Constraint Reuse

Constraint Reuse identifies constraints and their corresponding values that can be reused in regression testing the new version of the SUT. This is achieved through comparing the def-use chain of each variable present in the test paths of original and modified versions of the application. If the def-use of the variable remains same in the new version of the SUT, then the input test value of the variable can be reused in regression testing.

The process of collecting constraints and their input values, and determining reusability in the new version of the SUT, involves the following steps:

1. First, the set of test paths for the original and the modified version of the SUT are obtained from the *Path Generator*.
2. Next, the two set of test paths (original and modified versions) are compared to find the same variables that are used in both versions. Note that the input values for the old version already exist.
3. For each variable identified in the previous step, reusable constraint values for the new paths are collected by analyzing the variable's definitions and uses. The variable's definition is symbolically evaluated.

These steps are presented in Algorithms 1 and 2. Algorithm 1 returns an array of variables that have reusable constraint values.

Algorithm 1 takes four inputs: old test paths, new test paths, old PDGs, and new PDGs. 'Old' indicates the previous version, and 'new' refers to the current version being tested. The output of the

Algorithm 1 Find Variables with Reusable Input Values

1: **Inputs:**

oldPDG[]: An array of PDGs for the original version.

newPDG[]: An array of PDGs for the modified version.

oldPath[]: An array of identified test paths for the original version.

newPath[]: An array of identified test paths for the modified version.

2: **Outputs:**

variablesWithReusableInputValues: An array of variables that have same constraints in the original and modified version, and hence the test input value for these variables can be reused.

3: **Declare:**

oldDefUseMap: An array of all variables and their corresponding mapping to def-use, block numbers, code statements in the original version.

newDefUseMap: An array of all variables and their corresponding mapping to def-use, block numbers, code statements in the modified version.

4: **procedure** GETVARIABLESWITHREUSABLEINPUTVALUES($oldPDG$, $newPDG$, $oldPath$, $newPath$)
5: $oldDefUseMap \leftarrow \emptyset$
6: $newDefUseMap \leftarrow \emptyset$
7: $variablesWithReusableInputValues \leftarrow \emptyset$
8: **for** $n \longleftarrow 0$, $n < newPath$.size(), n++ **do**
9: $currentBlock \leftarrow newPDG.getBlock(newPath.getBlockID(n))$
10: **if** $currentBlock.HasDefOrUse()$ **then**
11: UpdateDefUseMap(currentBlock, newDefUseMap)
12: **end if**
13: **end for**
14: **for** $n \longleftarrow 0$, $n < oldPath$.size(), n++ **do**
15: $currentBlock \leftarrow oldPDG.getBlock(oldPath.getBlockID(n))$
16: **if** $currentBlock.HasDefOrUse()$ **then**
17: UpdateDefUseMap(currentBlock, oldDefUseMap)
18: **end if**
19: **end for**
20: **for** $n \longleftarrow 0$, $n < newDefUseMap$.size(), n++ **do**
21: $currentVar \leftarrow newDefUseMap.getVars(n)$
22: **if** $oldDefUseMap.getVars.contains(currentVar)$ **then**
23: $isSimilar \leftarrow CheckSimilarity(currentVar, oldDefUseMap, newDefUseMap)$
24: **if** $isSimilar$ **then**
25: $variablesWithReusableInputValues$.Add($currentVar$)
26: **end if**
27: **end if**
28: **end for**
29: **end procedure**

algorithm is a list of reusable variables. Line 3 declares two maps corresponding to the old and new versions of the application. The map contains the def-use chain for each variable that appears in the test paths. Mapping information includes the PDG block number of the variable's definition, variable's information such as name, data type, and the corresponding PDG nodes where the variable is used. Lines 8 through 13 show steps for preparing this map for the new version of an application. This loop iterates through each node in the new PDGs. If a block in the node has either a definition or a use of a variable, then function

'*UpdateDefUseMap*' is invoked to build the map for that variable. Similarly, lines 14 through 19 prepare def-use mapping for old test paths. Lines 20 through 28 compare the mapping information corresponding to the old and new versions of the application to determine reusable variables. This process loops through each map in '*newDefUseMap*' (line 20) and checks if the variable corresponding to the map also occurs in '*oldDefUseMap*' (line 21). If the variable is present, then the variables are compared for similarity by invoking function '*CheckSimilarity*' (lines 22 and 23). If this function returns '*true*', then the variable is added to '*variablewithReusableInputValues*'. Otherwise, the variable is discarded.

Algorithm 2 Check Similarity

1: **Inputs:**

currentVar: A variable and its corresponding mapping to def-use, block numbers and code statements in the new version.

oldDefUseMap: An array of all variables and their corresponding mapping to def-use, block numbers, code statements in the original version.

newDefUseMap: An array of all variables and their corresponding mapping to def-use, block numbers, code statements in the modified version.

2: **Outputs:**

true or *false*. A boolean value indicating if the given variable has similar def-use in the old and new versions.

3: **Declare:**

oldPathStatements: An array of code statements that either have definitions or uses of the given variable in the old version.

newPathStatements: An array of code statements that either have definitions or uses of the given variable in the new version.

4: **procedure** CHECKSIMILARITY($currentVar$, $oldDefUseMap$, $newDefUseMap$)
5: $oldPathStatements \leftarrow oldDefUseMap.getStmts(currentVar)$
6: $newPathStatements \leftarrow newDefUseMap.getStmts(currentVar)$
7: **for** $n \longleftarrow 0$, $n < newPathStatements$.size(), n++ **do**
8: **if** (CompleteMatch(newPathStatements[n], oldPathStatements[n]) $\neq true$) **then**
9: **if** (PartialMatch(newPathStatements[n], oldPathStatements[n]) $\neq true$) **then**
10: return *false*
11: **end if**
12: **end if**
13: $isVariableDependent \leftarrow CheckVariableDependency(newPathStatements[n], oldPathStatements[n])$
14: **if** $isVariableDependent$ **then**
15: $isResolved \leftarrow ResolveVariableDependency(newPathStatements[n], oldPathStatements[n])$
16: **if** $\neg isResolved$ **then**
17: return *false*
18: **end if**
19: **end if**
20: **end for**
21: return *true*
22: **end procedure**

Algorithm 2 shows the procedure for checking variable similarity. The inputs for this algorithm are mapping information of the variable, '*oldDefUseMap*' and '*newDefUseMap*' (line 1), and output is a Boolean value, where '*true*' indicates that the variables in the old and new versions of the application are similar, and *false* indicates otherwise (line 2). To begin the process, two data structures are declared, '*oldPathStatements*' and '*newPathStatements*', to store the actual code statements of the variables in old and new versions of the application (line 3). Lines 7 through 20 iterate through each code statement in '*newPathStatements*' comparing with the corresponding statement from '*oldPathStatements*'. If the statements do not match perfectly (line 8), then a partial match is checked. If the statements do not have a partial match (line 9), then this procedure returns *false* and exits. If either a perfect or a partial match is found, then the algorithm continues to find variable dependency. Variable dependency occurs if some parts of the new and old statements perfectly match, but there are other variable(s) that do not match. In such cases, the def-use of mismatched or dependent variables needs to be checked for similarity.

Suppose that a statement in the old version, '*if* $(a \leq 12)$', is changed to '*if* $(a \leq 12 || b \geq 7)$'. This would result in a partial match for the condition '$a \leq 12$'. If the def-use of variable 'a' remains the same across the versions, then the additional condition to satisfy for variable 'a' to be considered reusable is the def-use of variable 'b', which is the dependent variable, should remain unchanged between the versions. '*CheckVariableDependency*' at line 13 checks if there are such dependent variables and if so, '*ResolveVariableDependency*' at line 15 verifies if the def-use of the dependent variables remains the same between versions. If def-use of the dependent variables is found to be different, then the procedure returns '*false*' and exits. Otherwise, the procedure returns '*true*' confirming that the def-use of dependent variables has remained the same.

3.2.1.1. An Example of Reusable Input Variable Identification

The process of identifying variables with reusable constraint values is explained through an example in this section.

Consider the two pairs of consecutive versions v0.php, v1.php and v1.php, v2.php of a simple PHP program shown in Figure 3.2 and Figure 3.3, respectively. Modified statements in *v1* and *v2* are at lines 8 and 7, respectively, as commented in the code. In *v1*, line 8 changed from '*$a = $a - 1*' to '*$a = $a -3*'. Line 7 in *v2* changed from '*$b = 5*' in *v1* to '*$b = $b -1*'.

v0.php (Original Version)

```
function foo($a = 0, $b = 0)
{
    echo "Value of a is $a";
    echo "Value of b is $b";
    if ($a < 12)
    {
        $b = 5;
        $a = $a - 1;
    }
    else
    {
        $b = $b + 3;
    }
    if ($a > 7)
    {
        if ($b == 5)
        {
            echo "a";
        }
        else
        {
            $b = 7;
        }
    }
    echo "b";
    echo "Done Processing.";
}
```

v1.php (Modified Version)

```
1  function foo($a = 0, $b = 0)
2  {
3      echo "Value of a is $a";
4      echo "Value of b is $b";
5      if ($a < 12)
6      {
7          $b = 5
8          $a = $a - 3; // CHANGED
9      }
10     else
11     {
12         $b = $b + 3;
13     }
14     if ($a > 7)
15     {
16         if ($b == 5)
17         {
18             echo "a";
19         }
20         else
21         {
22             $b = 7;
23         }
24     }
25     echo "b";
26     echo "Done Processing.";
27 }
```

Figure 3.2. Two Consecutive Versions (*v0* and *v1*) of a PHP Source File.

One of the regression test paths for *v1* is {1, 2, 3, 4, 5, 6, 7, 8, 14, 15, 16, 17, 18, 25, 26}. In this path, there are three constraints, and those are '*$a <12*', '*$a - 3 >7*' and '*$b = 5*'. Before solving these constraints, it is checked if the constraints have remained unchanged between the original and modified versions. Constraints '*$a <12*' and '*$b = 5*' have remained the same in *v1*. Line 8 in *v1* modified the definition of variable '*a*', and thus the symbolic value of variable '*a*' is different *v0*. This means that the existing value of variable '*a*' in a test used in *v0* may no longer solve the constraint '*$a - 3 >7*'. For instance, if the existing input value of variable '*a*' is 9, then this value satisfies constraints '*$a <12*' and '*$a - 1 >7*' but not '*$a - 3 >7*'. The def-use of variable '*b*' has remained the same between *v0* and *v1*. A value of 5 for variable '*b*' will solve the constraint '*$b = 5*'. Therefore, variable '*a*' is not a reusable variable, while '*b*' is a reusable input variable.

v1.php (Original Version)

```
function foo($a = 0, $b = 0)
{
    echo "Value of a is $a";
    echo "Value of b is $b";
    if ($a < 12)
    {
        $b = 5
        $a = $a - 3;
    }
    else
    {
        $b = $b + 3;
    }
    if ($a > 7)
    {
        if ($b == 5)
        {
            echo "a";
        }
        else
        {
            $b = 7;
        }
    }
    echo "b";
    echo "Done Processing.";
}
```

v2.php (Modified Version)

```
1  function foo($a = 0, $b = 0)
2  {
3      echo "Value of a is $a";
4      echo "Value of b is $b";
5      if ($a < 12)
6      {
7          $b = $b - 1; // CHANGED
8          $a = $a - 3;
9      }
10     else
11     {
12         $b = $b + 3;
13     }
14     if ($a > 7)
15     {
16         if ($b == 5)
17         {
18             echo "a";
19         }
20         else
21         {
22             $b = 7;
23         }
24     }
25     echo "b";
26     echo "Done Processing.";
27 }
```

Figure 3.3. Two Consecutive Versions (*v1* and *v2*) of a PHP Source File.

Next, consider versions *v1* and *v2* shown in Figure 3.3. The modified statement between these versions is line 7, where '*$b = 5*' changed to '*$b = $b - 1*'. A regression test path for this version pair is {1, 2, 3, 4, 5, 6, 7, 8, 14, 15, 16, 17, 18, 25, 26}, and constraints for this path are '*$a <12*', '*$a - 3 >7*' and '*$b - 1 = 5*'. Constraint '*$a <12*' remained the same across all three versions. Constraint '*$a - 3 >7*' remained same between *v1* and *v2*. In *v2*, the symbolic value of variable '*b*' is modified due to the code change at line 7. This means that the existing value of variable '*b*' in a test used in *v1* may no longer solve the constraint '*$b - 1 = 5*'. For example, if the input value for variable '*b*' is 5 in *v1*, then this value will be modified to 4 at line 7 in *v2*, and thus at line 16 the condition '*$b = 5*' returns *false*. Therefore, the value of variable '*b*' from *v1* is not reusable in *v2* while variable '*a*' is reusable.

This example does not include cases that assign variables to other variables or non-primitive variables, but a process like the one shown in our example is applied to such variables. However, in the case of non-primitive variables, before applying the process, they are tokenized in our preprocessing step (conversion from PHP code to AST and HIR [25]).

3.2.2. Test Selection

As shown in Algorithm 3, *Test Selection* selects a subset of tests from the existing tests that are applicable to the current version using the *DejaVu* approach, and identifies reusable constraint values using the procedure described in *Constraint Reuse*, which automates the process of identifying obsolete tests (tests that include input values that are not applicable to the modified version).

The inputs for this procedure are the new and old PDGs, instrumented source code of the old version, slices identified by analyzing the changes between two versions, and tests for the old version. The output is '*SelectedTestsWithValues*', an array of selected tests with reusable input values. The algorithm invokes '*TestHistoryBuilder*' and '*SelectExistingTests*' functions to obtain an array of selected tests that cover the modified code. Procedures for these two functions are shown in Algorithms 4 and 5. Once the tests are selected from the existing tests, the algorithm invokes function '*GetVariablesWithReusableInputValues*' (Algorithm 1), which was explained in the previous subsection, to find reusable input values for the selected tests.

The test history builder (THB) procedure shown in Algorithm 4 reads PDGs, instrumented source code, and tests for the old version. It returns '*TestHistory*', a data structure containing node coverage for each test. THB uses PHPUnit [73] to execute each existing unit test against the instrumented version of the source code to collect test history information. *PDG Generator*, in addition to generating PDGs, also generates instrumented code, which enables THB to build test history. Test history captures node coverage, which in this case is a list of PDG nodes and the corresponding number of times the nodes were covered during test execution.

The '*SelectExistingTests*' procedure shown in Algorithm 5 uses test history, PDGs for the old version, and program slices to determine the subset of tests that cover modified code paths in the new version of the application. PDGs and slices are obtained from *Impact analysis*. A slice is a list of code statements or blocks (See our previous work [25] for more details about slices.). Each slice is a code path that includes the modified block and impacted blocks. When selecting tests, we check if a test covers a path listed in the slice.

Algorithm 3 Test Selection

Inputs:

oldPDG[]: An array of PDGs for the original version.

newPDG[]: An array of PDGs for the modified version.

Inst[]: An array of instrumented code of each source file from the original version.

Slices[]: An array of slices identified by analyzing the changes that happened between the original and modified versions.

Tests[]: An array of existing tests of the original version.

Outputs:

SelectedTestsWithValues: An array of selected tests with reusable constraint values for the modified version.

Declare:

oldPath[]: An array of test paths in the original version.

newPath[]: An array of test paths in the modified version.

SelectedExistingTests[]: An array of existing tests that cover the modified code paths in the new version.

TestHistory[]: An array of existing tests and the corresponding PDG block coverage.

VariablesWithReusableInputValues[]: An array of input variables that have same def-use in the new version.

procedure TESTSELECTION($oldPDG$, $newPDG$, $Inst$, $Slices$, $Tests$)
  $SelectedTestsWithValues \leftarrow \emptyset$
  $TestHistory \leftarrow TestHistoryBuilder(oldPDG, Inst, Tests)$
  $SelectedExistingTests \leftarrow SelectExistingTests(TestHistory, oldPDG, Slices)$
  $VariablesWithReusableInputValues \leftarrow GetVariablesWithReusableInputValues($
    oldPath,newPath,oldPDG,newPDG)
  for $n \longleftarrow 0$, $n < SelectedExistingTests$.size(), n++ **do**
    $IsTestReusable \leftarrow true$
    for $v \longleftarrow 0$, $v < SelectedExistingTests[n]$.GetInputParameterCount(), v++ **do**
      $InputParameter \leftarrow SelectedExistingTests$[n].$GetInputParameter$(v)
      if $InputParameter \notin VariablesWithReusableInputValues$ **then**
        $IsTestReusable \leftarrow false$
        break
      end if
    end for
    if $IsTestReusable$ **then**
      $SelectedTestsWithValues$.add($SelectedExistingTests$[n])
    end if
  end for
  return $SelectedTestsWithValues$
end procedure

Algorithm 4 Test History Builder

procedure TESTHISTORYBUILDER($oldPDG$, $Inst$, $Tests$)
  for $t \longleftarrow 0$, $t < Tests$.size(), t++ **do**
    $coverage \leftarrow phpunit$(Inst,Test[$n$])
    for $block \longleftarrow 0$, $block < coverage$.size(), $block$++ **do**
      if $coverage$.GetBlock($block$) $\in oldPDG$ **then**
        $TestHistory$.add($Test[t]$, $coverage$.GetBlock($block$))
      end if
    end for
  end for
  return $TestHistory$
end procedure

Algorithm 5 SelectExistingTests

```
1:  procedure SELECTEXISTINGTESTS(TestHistory, oldPDG, Slices)
2:      for s ⟵ 0, s < Slices.size(), s++ do
3:          blocks ← Slices[s].GetBlocks()
4:          pdgBlocks ← oldPDG.GetBlocks(blocks)
5:          for history ⟵ 0, history < TestHistory.size(), history++ do
6:              if TestHistory[history].GetBlocks() ∈ pdgBlocks then
7:                  SelectedExistingTests.add(TestHistory[history].GetTest())
8:              end if
9:          end for
10:     end for
11:     return SelectedExistingTests
12: end procedure
```

In PARTE, the PDGs of the old and new versions are compared to determine the statements that were added, edited, or deleted. Slicer uses the modification information to calculate the code change impact set. For the variables defined in a statement that was modified, forward slicing is performed by finding all statements that are data dependent on such variables. Next, backward slicing is performed to locate all statements on which modified statement is data dependent. Forward and backward slices are combined into a single slice for each such modified statement. Lines 2 through 10 in Algorithm 5 iterate through each slice, checking test history to see whether all PDG blocks in the slice are covered by any of the existing tests. If any such tests are found, they are added to the '*SelectedExistingTests*' array.

3.2.2.1. An Example of Test Selection

Test selection is explained through an example. Consider the same three versions shown in Figure 3.2 and Figure 3.3. The unit tests for v0 are shown in Figure 3.4. The test paths that are expected to be covered by the unit tests are shown in the comment lines. For example, '*test1Foo*' covers a test path represented by the following set of statement numbers: {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 25, 26, 27}.

v0unittest.php

```
1  class FooTests extends PHPUnit_Framework_TestCase{
2  // Test Path {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 25, 26, 27}
3  public function test1Foo(){
4     echo "Input value for variable a is not provided.";
5     echo "Input value for variable b is not provided.";
6     foo();}
7  // Test Path {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 25, 26, 27}
8  public function test2Foo(){
9     echo "Input value for variable a is 8.";
10    echo "Input value for variable b is 0.";
11    foo(8,0);}
12 // Test Path {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 15, 16, 17, 18, 19, 24, 25, 26, 27}
13 public function test3Foo(){
14    echo "Input value for variable a is 9.";
15    echo "Input value for variable b is 5.";
16    foo(9,5);}
17 // Test Path {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 15, 16, 20, 21, 22, 23, 24, 25, 26, 27}
18 public function test4Foo(){
19    echo "Input value for variable a is 11.";
20    echo "Input value for variable b is 6.";
21    foo(11,6);}
22 // Test Path {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 15, 16, 17, 18, 19, 24, 25, 26, 27}
23 public function test5Foo(){
24    echo "Input value for variable a is 11.";
25    echo "Input value for variable b is 5.";
26    foo(11,5);}
27 // Test Path {1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15, 16, 20, 21, 22, 23, 24, 25, 26, 27}
28 public function test6Foo(){
29    echo "Input value for variable a is 12.";
30    echo "Input value for variable b is 0.";
31    foo(12,0);}
32 // Test Path {1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 24, 25, 26, 27}
33 public function test7Foo(){
34    echo "Input value for variable a is 12.";
35    echo "Input value for variable b is 2.";
36    foo(12,2);}
37 }
```

Figure 3.4. Unit Tests for v0.php (Original Version).

THB runs these unit tests against source code to collect node and block coverage information. '*SelectExistingTests*' reads this coverage information to identify tests that cover paths with code changes. In this example, PARTE's impact analysis lists such paths in *v1*. '*SelectExistingTests*' identifies tests '*test1Foo*', '*test2Foo*', '*test3Foo*', and '*test4Foo*' as test that cover paths containing code changes in *v1*.

Although the selected tests cover the code modifications, the actual input values may not be applicable to the new version depending on the type of modification in the new source code. For the version pair *v0* and *v1*, variable '*b*' is unchanged with respect to def-use while that of variable '*a*' has changed. This means tests that pass a value to variable '*a*' cannot be reused without manually inspecting whether the structural coverage of the test is maintained in *v1*. For example, '*test3Foo*' is designed to cover a test path represented by the following set of line numbers: {1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 15, 16, 17, 18, 19, 24, 25, 26, 27} in *v0*. In *v1*, due to change in definition of variable '*a*' from '*$a-1*' to '*$a-3*' the condition at line 14 may return false depending on the value of variable '*a*', and execution will not cover lines 15 through 24. Suppose that a value of 10 for variable '*a*' will maintain the path in *v0* and *v1*, but a value of 9 will execute the path in *v0* but not in *v1* because '*$a-3*' will not be greater than 7, which is required for the condition at line 14 to return *true*. This means the input value for variable '*a*' in '*test3Foo*' needs to be manually verified to check whether the current value provided for this variable allows the test to cover the test path in *v1*. If the value does not cover the test path, then the test is obsolete. Thus, test case '*test3Foo*' cannot be selected, and the final set of selected regression tests for *v1* are '*test1Foo*', *test2Foo*, and *test4Foo*. Similarly, between *v1* and *v2*, variable '*b*' does not have the same def-use. Thus, for regression test of *v2*, only '*test4Foo*' is selected.

3.3. Test Repair

As described in Section 3.2.1, *Reusable Tests* not only identifies existing tests that can be reused for the new version but also identifies tests that are obsolete. Though the test repair approaches described in literature, are shown to be effective, we identified a few limitations. First, when the number of obsolete tests is large, a developer should be provided guidance on which of the obsolete tests are likely to cover the code paths that have undergone a change in the modified version of the application. Since there is a cost associated with repairing a test, it is efficient to do a selective repair as opposed to a repair all tests approach. Second, use multiple strategies to repair a test. Not only fixing the failing assertions but also the failures introduced due to method signature changes should be incorporated into the repair technique [63] In POTR, we chose to implement two most common types of test repairs, which fix assertions and update method invocations. To perform such repairs, using the PARTE approach, PDGs of the source and test code are analyzed to identify

the assertions and method invocations that require fixes. *Test Repair* component performs such repairs on the obsolete tests. As shown in Figure 3.1, *Test Repair* consists of two components: MSA (Method Signature Analyzer) and AA (Assertion Analyzer). MSA repairs obsolete tests by detecting changes to the parameters in the method signature that caused a test to become obsolete. AA repairs failing assertions in an obsolete test. We describe these two components in detail, including algorithms and an example.

3.3.1. Method Signature Analyzer (MSA)

MSA attempts to repair an obsolete test by identifying the change in the signature of the method covered by that test. MSA compares the PDG node corresponding to the method in the original and the modified versions. The PDG node provides the details of parameters in the method's signature. These details include the name of the parameter, data type and default value. MSA compares these details to identify if a parameter in the method signature was added/removed, or its data type was modified.

The overview of MSA is as follows:

1. Identify the method whose invocation in the test needs to be updated. Test failure messages contain the method name and line number(s) where the invocation caused a failure.

2. Gather the method's parameter details from PDGs of the original and modified versions of SUT.

3. Compare the parameter details between the two versions to determine the changes.

These steps are shown in Algorithms 6 and 7. Algorithm 6 returns an array that contains method parameter analysis results for each obsolete test.

Algorithm 6 takes three inputs: obsolete tests, PDG for the original version, and PDG for the modified version. The output of the algorithm is an array of suggestions determined based on method parameter comparison analysis results. Line 3 declares an array that will hold the repair suggestions for obsolete tests. Lines 4 through 13 iterate over each obsolete test. Line 7 prepares the test object for the obsolete test in the current iteration. For this test, line 8 extracts the fully qualified name of the method that is covered by the test. Lines 9 and 10 extract the method's parameter details, which includes name, data type and default value. These details are obtained from the method's PDG node. Line 11 invokes '*CompareParameters*' procedure that performs the comparison of the method's parameters between the versions to identify the changes. Line 12 captures the comparison analysis for the test in the current iteration. After iterating through all obsolete tests, line 14 returns the analysis of each obsolete test as repair suggestions.

Algorithm 6 Method Signature Analyzer

1: **Inputs:**

obsoleteTests[]: An array of obsolete tests for the modified version.

oldPDG[]: An array of PDGs for the original version.

newPDG[]: An array of PDGs for the modified version.

2: **Outputs:**

RepairedTests: An array containing repair suggestions for obsolete tests.

3: **Declare:**

RepairedTests[]: An array containing repair suggestions for obsolete tests.

4: **procedure** METHODSIGNATUREANALYZER($obsoleteTests$, $oldPDG$, $newPDG$)
5: $RepairedTests \leftarrow \emptyset$
6: **for** $n \longleftarrow 0$, $n <$ *obsoleteTests*.size(), n++ **do**
7: $test \leftarrow obsoleteTests[n]$
8: $testMethod \leftarrow test.GetTestMethod()$
9: $oldParams \leftarrow oldPdg.GetBlocks(testMethod).GetParameters()$
10: $newParams \leftarrow newPdg.GetBlocks(testMethod).GetParameters()$
11: $paramStatus \leftarrow CompareParameters(oldParams, newParams)$
12: $RepairedTests.Add(test, paramStatus)$
13: **end for**
14: **return** $RepairedTests$
15: **end procedure**

Algorithm 7 Compare Method Parameters

1: **procedure** COMPAREPARAMETERS($oldParams$, $newParams$)
2: $paramAnalysis \leftarrow \emptyset$
3: $paramStatus \leftarrow \emptyset$
4: **for** $i \longleftarrow 0$, $i < newParams$.size(), i++ **do**
5: $newParameter \leftarrow newParams[i]$
6: **if** $newParameter \notin oldParams$ **then**
7: $paramStatus \leftarrow$ ("Parameter *newParameter.Name* was added.")
8: **else**
9: **for** $j \longleftarrow 0$, $j <$ *oldParams*.size(), j++ **do**
10: $oldParameter \leftarrow oldParams[j]$
11: **if** $oldParameter \notin newParams$ **then**
12: $paramStatus \leftarrow$ ("Parameter *oldParameter.Name* was removed.")
13: **else**
14: **if** $oldParameter.Name \equiv newParameter.Name$ AND $oldParameter.Type \not\equiv newParameter.Type$ **then**
15: $paramStatus \leftarrow$ ("Parameter data type of *oldParameter.Name* was modified from oldParameter.Type to newParameter.Type")
16: **end if**
17: **end if**
18: **end for**
19: **end if**
20: $paramAnalysis.Add(paramStatus)$
21: **end for**
22: **return** $paramAnalysis$
23: **end procedure**

Inputs for '*CompareParameters*' procedure are two arrays: '*oldParams*' and '*newParams*', which contain parameter details of a method in the original and modified version of SUT, respectively. Lines

4 through 21 iterate over each parameter in '*newParams*'. Line 6 checks if a parameter in the modified version of the method does not exist in the original version, and if so the status of this parameter is set as added. Lines 9 through 18 iterate over each parameter in the original version of the method. Line 11 checks if a parameter in the original version of the method does not exist in the modified version, and if so the status of the parameter is set as removed. Line 14 checks if the data type of a parameter has remained the same between the versions. If not, then the parameter status is set as data type modified. After both the inner and outer loops finish, comparison analysis result is returned to '*paramStatus*' at line 12 of Algorithm 6.

3.3.1.1. An Example for Method Signature Analyzer

In this section, the procedure for MSA is explained through an example. Consider the two consecutive versions of '*findShortestPathCode*' function from *Composer*, which is one of the applications used in our experiments. Figure 3.5 and Figure 3.6 show version 1.0.0 and 1.1.0 of '*findShortestPathCode*' function, respectively.

findShortestPathCode (Original Version)

```
1  public function findShortestPath($from, $to, $directories = false) {
2  if (!$this->isAbsolutePath($from) || !$this->isAbsolutePath($to)) {
3      throw new \InvalidArgumentException();
4  }
5  $from = lcfirst($this->normalizePath($from));
6  $to = lcfirst($this->normalizePath($to));
7  if ($directories) {
8      $from = rtrim($from, '/') . '/dummy_file';
9  }
10 if (dirname($from) === dirname($to)) {
11     return './'.basename($to);
12 }
13 $commonPath = $to;
14 while (strpos($from.'/', $commonPath.'/') !== 0
15     && '/' !== $commonPath && !preg_match('{^[a-z]:/?$}i', $commonPath)) {
16     $commonPath = strtr(dirname($commonPath), '\\', '/');
17 }
18 if (0 !== strpos($from, $commonPath) || '/' === $commonPath) {
19     return $to;
20 }
21 $commonPath = rtrim($commonPath, '/') . '/';
22 $sourcePathDepth = substr_count(substr($from, strlen($commonPath)), '/');
23 $commonPathCode = str_repeat('../', $sourcePathDepth);
24 return ($commonPathCode . substr($to, strlen($commonPath))) ?: './';}
```

Figure 3.5. Function *findShortestPathCode* in Version 1.0.0 of *Composer*.

findShortestPathCode (Modified Version)

```
1  public function findShortestPathCode($from, $to, $directories = false, $staticCode = false) {
2  if (!$this->isAbsolutePath($from) || !$this->isAbsolutePath($to)) {
3      throw new \InvalidArgumentException();
4  }
5  $from = lcfirst($this->normalizePath($from));
6  $to = lcfirst($this->normalizePath($to));
7  if ($from === $to) {
8      return $directories ? '__DIR__' : '__FILE__';
9  }
10 $commonPath = $to;
11 while (strpos($from.'/', $commonPath.'/') !== 0 && '/' !== $commonPath
12         && !preg_match('{^[a-z]:/?$}i', $commonPath) && '.' !== $commonPath) {
13     $commonPath = strtr(dirname($commonPath), '\\', '/');
14 }
15 if (0 !== strpos($from, $commonPath) || '/' === $commonPath
16     || '.' === $commonPath) {
17     return var_export($to, true);
18 }
19 $commonPath = rtrim($commonPath, '/') . '/';
20 if (strpos($to, $from.'/') === 0) {
21     return '__DIR__ . '.var_export(substr($to, strlen($from)), true);
22 }
23 $sourcePathDepth = substr_count(substr($from, strlen($commonPath)), '/') + $directories;
24
25 // CHANGED
26 if ($staticCode) {
27     $commonPathCode = "__DIR__ . '".str_repeat('/..', $sourcePathDepth)."'";
28 } else {
29     $commonPathCode = str_repeat('dirname(', $sourcePathDepth).'__DIR__'.str_repeat(')',
30                         $sourcePathDepth);
31 }
32
33 $relTarget = substr($to, strlen($commonPath));
34 return $commonPathCode . (strlen($relTarget) ? '.' . var_export('/' . $relTarget, true) : '');}
```

Figure 3.6. Function *findShortestPathCode* in Version 1.1.0 of *Composer*.

'*findShortestPathCode*' function is a file system utility that returns the PHP code that when executed in the '*$from*' path, will return the path to '*$to*'. Observe that in version 1.1.0, the function signature has a new parameter called '*staticCode*', and this parameter gets used in the newly introduced '*if-else*' block at lines 25 through 31. These changes were part of performance improvements [17]. Figure 3.7 shows the PDG block corresponding to the function signature in version 1.1.0. Observe that the parameter details, which includes the name, type and default value are available at lines 11 through 14. Figure 3.8 shows a unit test for '*findShortestPathCode*' function in version 1.0.0.

Observe that the parameter '*$from*' in '*findShortestPathCode*' does not have the same definition and use between versions 1.0.0 and 1.1.0. Value of '*$from*' is assigned to another variable ('*sourcePathDepth*'), see line 23 in version 1.1.0. The use of '*sourcePathDepth*' changed due to the introduction of the *if-else*

PDG for *findShortestPathCode* (Modified Version)

```
<PDGNode BlockID="256" ContextString="def1methodDec256"
        ParentContext="def1" type="ClassMethodDec">
<Exits>
   <Exit BlockID="257"/>
</Exits>
<Entries>
   <Entry BlockID="232"/>
</Entries>
<MethodCallExits/>
<Defs>
   <Def Name="$from" Type="" Value=""/>
   <Def Name="$to" Type="" Value=""/>
   <Def Name="$directories" Type="" Value="False"/>
   <Def Name="$staticCode" Type="" Value="False"/>
<Defs>
<Uses/>
<DataPreds/>
<DataDeps/>
<CDChildren>
   <!−− List of children −−>
</CDChildren>
<StatementValue>
   public findShortestPathCode($from, $to, $directories=False, $staticCode=False)
</StatementValue>
</PDGNode>
```

Figure 3.7. PDG for One of the Modified Blocks in *findShortestPathCode*.

testFindShortestPathCode (Original Version)

```
public function testFindShortestPathCode($a, $b, $directory, $expected) {
   $fs = new Filesystem;
   $this−>assertEquals($expected, $fs−>findShortestPathCode($a, $b, $directory));
}
```

Figure 3.8. A Unit Test for *findShortestPathCode*.

block in version 1.1.0. Hence, due to these changes the unit test shown in Figure 3.8 is identified as an obsolete test and this test will be included in the list of obsolete tests passed to MSA for repair. As described in Algorithm 6, MSA extracts the parameter details of '*findShortestPathCode*' from the PDG nodes corresponding to the original and modified versions, compares and identifies the newly added parameter, which is '*$staticCode*' and its default value as '*false*'. MSA gathers this analysis and creates a copy of the test with the identified repairs to the method signature. Figure 3.9 shows the repaired version of the test. Providing such repaired copy of a test can assist developers to assess the coverage of existing test and determine if a new test, for example, with '*$staticCode*' value as '*true*' would be valuable to include in the regression test suite.

testFindShortestPathCode (Repaired Version)

```
1  public function testFindShortestPathCode($a, $b, $directory, $expected, $static = false) {
2      $fs = new Filesystem;
3      $this->assertEquals($expected, $fs->findShortestPathCode($a, $b, $directory, $static));
4  }
```

Figure 3.9. Repaired Unit Test for *findShortestPathCode*.

3.3.2. Assertion Analyzer (AA)

AA attempts to repair an obsolete test by updating the test oracle. AA identifies the new value for the test oracle, which we refer to as the assertion, based on the PHPUnit failure message returned from the test execution. The steps below describe how AA performs test repair.

1. AA uses PHPUnit to execute the identified obsolete test. AA parses the test failure message to identify the new value of the test oracle.

2. If the expected object in the assertion method of the test is a constant, which is determined from the PDG node corresponding to the assertion statement in the test, then a copy of the original test is created. In the copy, the excepted value in the assertion method is replaced with the new value that was captured in the previous step.

3. Similar to the previous step, if the expected object in the assertion is a variable, then in the copy of test, the variable's last definition before the assertion is updated with the new value. PDG of the test is used to locate the statement that corresponds to the variable's definition.

4. Finally, AA uses PHPUnit to execute the newly created copy of the test. If the assertion succeeds, then the new test is presented to the user as the repaired version of the obsolete test.

These steps are shown in Algorithm 8. The Input for Algorithm 8 is an array of obsolete tests that were identified using *Reusable Tests* component. This algorithm returns an array that contains the corresponding repaired copy of each obsolete test. Lines 6 through 25 iterate over each obsolete test. Line 8 captures the result from PHPUnit obtained when the obsolete test is executed against the modified version of SUT. Line 9 verifies that the test indeed fails. Line 12 parses the assertion failure message to identify the new value for the test oracle, which is the assertion in the test. The new value is assigned to '*repairValue*'. Line 14 extracts the PDG block corresponding to the assertion in the test. From this PDG block, it is possible to determine if a variable or a constant is used in the assertion. Line 16 invokes '*CreateTest*' method to create a

copy of the original test and updates the assertion with the value captured in '*repairValue*'. Similarly, line 19 creates a copy of the test but with the variable definition updated to '*repairValue*'. PDG of the test enables identification of the statement corresponding to the variable definition. Line 21 captures the execution result of the repaired test. Line 22 checks if the execution did indeed succeed, and if so, the repaired test is added to '*RepairedTests*' array. After looping through all the obsolete tests, the repaired tests are available in '*RepairedTests*'

Algorithm 8 Assertion Analyzer

```
1:  Inputs:
        obsoleteTests[]: An array of obsolete tests for the modified version.
2:  Outputs:
        RepairedTests: An array containing repaired copy of the obsolete tests.
3:  Declare:
        RepairedTests[]: An array containing repaired copy for obsolete tests.
4:  procedure ASSERTANALYZER(obsoleteTests)
5:      RepairedTests ← ∅
6:      for n ⟵ 0, n < obsoleteTests.size(), n++ do
7:          test ← obsoleteTests[n]
8:          currentTestExecutionResult ← test.Execute()
9:          if currentTestExecutionResult.IsSuccess then
10:             continue
11:         end if
12:         repairValue ← currentTestExecutionResult.NewValue
13:         testPDG ← GeneratePDG(test)
14:         assertBlock ← testPDG.GetAssertBlock()
15:         if assertBlock.ExpectedType ≡ Literal then
16:             newTest ← CreateTest(test, repairValue)
17:         end if
18:         if assertBlock.ExpectedType ≡ Variable then
19:             newTest ← CreateTest(test, repairValue, assertBlock))
20:         end if
21:         repairTestExecutionResult ← newTest.Execute()
22:         if repairTestExecutionResult.IsSuccess then
23:             RepairedTests.Add(newTest)
24:         end if
25:     end for
26:     return RepairedTests
27: end procedure
```

3.3.2.1. An Example for Assertion Analyzer

In this section, *Assertion Analyzer* is explained using an example. Figure 3.10 and Figure 3.11 show '*getPlatformPath*' function in *Filesystem.php* of *Composer* in version 1.0.0 and 1.1.0, respectively. This function returns a platform specific file URI path for the given input path. A bug in this function was

identified [16] and fixed [20]. The fix was at line 4, where '*file:///([a-z])/)i*' was replaced with '*file:///([a-z])/):?)i*', notice the additional '***:?***' in the regular expression pattern of '*preg_replace*' function. *Reusable Tests* selects all tests from version 1.0.0 that cover this modified statement. Figure 3.12 shows an example of such test. The test would be considered obsolete because the definition of the variable '*$path*' was modified due to the change at line 4 in version 1.1.0.

getPlatformPath (Original Version)

```
1  public static function getPlatformPath($path) {
2      if (Platform::isWindows()) {
3          $path = preg_replace(
4              '{^(?:file:///([a-z])/)}i',
5              'file://$1:/', $path);
6      }
7
8      return preg_replace('{^file://}i', '', $path);
9  }
```

Figure 3.10. Function *getPlatformPath* in Version 1.0.0 of *Composer*.

getPlatformPath (Modified Version)

```
1  public static function getPlatformPath($path) {
2      if (Platform::isWindows()) {
3          $path = preg_replace(
4              '{^(?:file:///([a-z]):?/)}i',  // CHANGED
5              'file://$1:/', $path);
6      }
7
8      return preg_replace('{^file://}i', '', $path);
9  }
```

Figure 3.11. Function *getPlatformPath* in Version 1.1.0 of *Composer*.

The procedure for AA begins with passing an array of obsolete tests. In this example, the array contains '*getPlatformPathTest*'. AA executes this test against version 1.1.0 using PHPUnit. The test fails with an error message, "*the expected value file:///c:/repos/composer/.git does not match the actual value, which is file://c:/repos/composer/.git.*" AA parses this test failure message and identifies the difference between expected and actual values. The difference in this case is on the *Windows* platform, the actual value will have one less '/' character compared to the expected value. From the PDG of the '*getPlatformPathTest*',

getPlatformPathTest (Original Version)

```
1  namespace Composer\Test\Util;
2  use Composer\Util\Filesystem;
3  use Composer\TestCase;
4
5  class FilesystemTest extends TestCase {
6      private $fs;
7
8      public function getPlatformPathTest() {
9          $this->fs = new Filesystem;
10         $expectedPath = 'file:///c:/repos/composer/.git'
11         $actualPath = 'file:///c:/repos/composer/.git'
12
13         $this->assertEquals($expectedPath, $fs->getPlatformPath(actualPath));
14     }
15 }
```

Figure 3.12. A Unit Test for *getPlatformPath* Function.

AA detects that the assertion statement has a variable '*$expectedPath*'. AA navigates the dependency graph to identify the statement with the most recent definition of '*$expectedPath*'. AA prepares a copy of the test with '*$expectedPath*' set to '*file:///c:/repos/composer/.git*'. Figure 3.13 shows the repaired test. AA executes this repaired test against version 1.1.0 and verifies if the test indeed succeeds, and if so, AA presents this copy of the test as the repaired test to the developer who can review the change.

getPlatformPathTest (Repaired Version)

```
1  namespace Composer\Test\Util;
2  use Composer\Util\Filesystem;
3  use Composer\TestCase;
4
5  class FilesystemTest extends TestCase {
6      private $fs;
7
8      public function getPlatformPathTest() {
9          $this->fs = new Filesystem;
10         $expectedPath = 'file:///c:/repos/composer/.git'
11         $actualPath = 'file:///c:/repos/composer/.git'
12
13         $this->assertEquals($expectedPath, $fs->getPlatformPath(actualPath));
14     }
15 }
```

Figure 3.13. Repaired Unit Test for *getPlatformPath* Function.

4. INFRASTRUCTURE

This section describes the infrastructure that was developed to conduct our experiments.

4.1. Application Acquisition

One of the key requirements to conduct our experiments was to acquire PHP web applications. To verify the proposed approaches, we needed access to the source code of multiple consecutive versions of real-world applications. For the test selection and test repair experiments, it is required that the applications have unit tests available. To evaluate the effectiveness of the proposed approaches, a changelog of the versions should be available to verify if all the real faults were detected by the tests selected through our proposed approach. Additionally, applications with active users and forums to clarify any questions we have, would assist in our experimentation. Based on these criteria, we identified several applications, and 10 of which are listed in Table 4.1.

Table 4.1. PHP Web Applications.

Application	**Description**	**Lines of Code**
Composer	A dependency manager. Used to declare, manage and install dependencies.	73673
FAQForge	A content management system focused on creating Frequently Asked Questions (FAQs), manuals and guides.	1775
Grav	A flat-file, web-based content management system.	43627
Laravel	A web application framework.	2681
log4hp	A logging framework.	143392
Mambo	A content management system.	136427
Manits	An issue tracking application.	181966
MediaWiki	A web-based application to build a wiki, which is website where users collaboratively modify content.	740547
osCommerce	A store-management application.	67920
phpScheduleIt	A reservation and scheduling application.	52683

In Table 4.1, “Application” is the name of the PHP web application, “Description” is a one-line description of the application, and “Lines of Code” is the average number of lines of code across the multiple versions of the application that we reviewed. We listed the “Lines of Code” metric to show that the applications consider were non-trivial, and lot of effort went into vetting the applications needed for experimentation.

We manually reviewed each version pair i.e., consecutive versions of an application. We identified the modifications between the versions, and traced each modification to a real fault or a feature request. We reviewed the coverage of existing tests. In case the existing tests did not cover the modifications or a core functionality, then we manually prepared unit tests to improve the coverage. We ran the tests and manually verified the results to ensure that the tests filled the gaps in the coverage.

4.2. Tools of PARTE

As shown in Figure 3.1, PARTE is a collection of components and each component is a collection of tools. A list of such tools is shown in Table 4.2.

Table 4.2. Tools in PARTE.

Tool	**Description**	**Lines of Code**
AST2HIR	Convert an AST to HIR.	2986
Constraint Collector	Collect constraints in the given test paths.	1063
Constraint Reuse	Identifies reusable constraints.	515
Impact Analysis	Identifies the impact set based on static impact analysis.	845
Includer	Searches for include and require statements, and replaces with appropriate code.	3594
Path Generator	Identifies the regression test paths.	4078
PDG Generator	Generate PDGs and instrumented code.	12570
Slicer	Identifies program slices.	853
Test History Builder	Identifies test coverage for the given test.	730
Test Repair	Repairs the given obsolete tests.	9741
Test Selection	Identifies existing tests that cover the given regression test paths.	857

In Table 4.2, "Tool" is the name of the tool, "Description" is a one-line description of the tool, and "Lines of Code" is the number of lines of code in the tool. All these tools were developed using Java. Generation of PDG requires considering various PHP constructs such as class, methods, statements, loops conditionals, operators and variables. Hence the lines of code for *PDG Generator* was 12,570. In case of *Test Repair*, we had to handle various types of assertions supported by PHP, and detect changes to parameters in a method signature. This added up to 9,741 lines of code for *Test Repair*.

To control the process and data flow between PARTE components, we prepared Perl scripts. Before running these scripts on an application, we ensure environment cleanliness by starting an empty environment

using '*env -i*' Unix shell command. We did not use any visualization or containerization technologies that would have offered a cleaner environment for running the scripts on an application.

4.3. Organization of Infrastructure

To support the process of controlled experiments, we organized a directory structure so that the experiments can be replicated consistently and reliably. This structure is shown in Figure 4.1, and Table 4.3 provides a description of these directories.

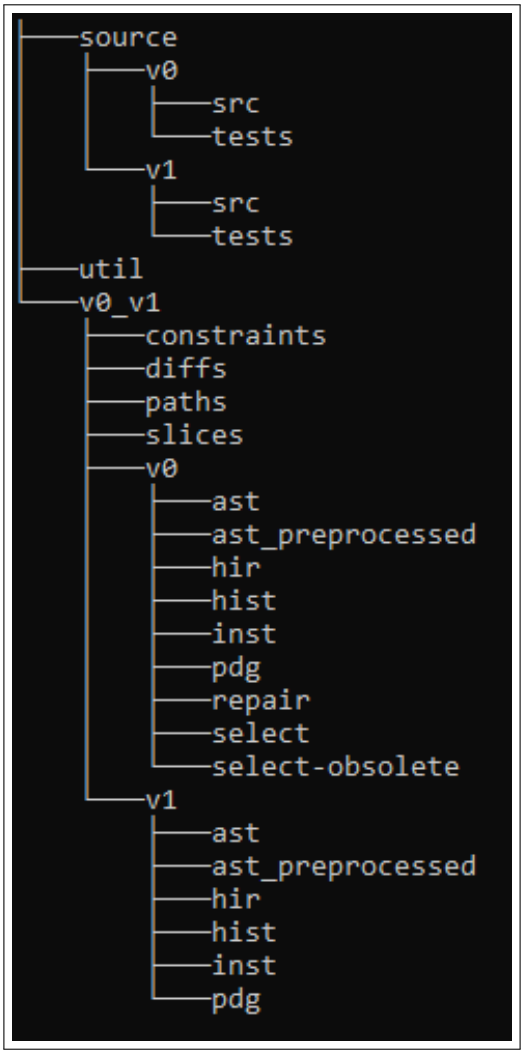

Figure 4.1. Directory Structure of the Infrastructure.

Items not listed in Figure 4.1 and Table 4.3 are the logs and the usage guides. These are essential in understanding the usage or the tools, output, and helpful in replicating the experiments. All the tools generate verbose logs, which are plain text files stored in the corresponding output folder. For example, logs

from PDG generation on original version (v0) of SUT will be under '*v0_v1/v0/pdg*'. A detailed usage guide was prepared for each tool. The guide can be accessed through a command line argument. For example, '*java -jar PDGGenerator.jar -help*' would display the usage guide for *PDGGenerator*. To keep track of the changes, we used *Apache Subversion*, which is an open-source software version control system.

Table 4.3. Description of Directories.

Directory	**Description**
source	Source code of the version pair of the SUT.
source/v0/src	Source code of the original version.
source/v0/tests	Unit tests for the original version.
source/v1/src	Source code of the modified version.
source/v1/tests	Unit tests for the modified version.
util	All the tools of PARTE. This includes the Java Archive (JAR) of each tool in PARTE, shell and Perl scripts.
v0_v1	Artifacts of running the tools over the source code.
v0_v1/constraints	Constraints in the regression paths.
v0_v1/diffs	Differences between the PDGs corresponding to the original and modified versions.
v0_v1/paths	Regression paths in the modified version.
v0_v1/v0/ast	ASTs for the original version.
v0_v1/v0/ast_preprocessed	ASTs corresponding to the preprocessed source code files of the original version.
v0_v1/v0/hir	HIR corresponding to the AST of the original version.
v0_v1/v0/inst	Instrumented source code of the original version.
v0_v1/v0/pdg	PDGs for the original version.
v0_v1/v0/select	Original version tests that are reusable for regression testing the modified version.
v0_v1/v0/select-obsolete	Original version tests that are identified as obsolete.
v0_v1/v0/repair	Obsolete tests that were successfully repaired.

5. EMPIRICAL STUDIES

To investigate the approaches described in Chapter 3, we performed a family of empirical studies. We describe each of them in the following subsections.

5.1. Experiment 1: Reusable Constraint Values

To assess the proposed approach that identifies reusable constraints values, we performed a controlled experiment. The design, data collection, analysis and reporting of the results of the experiments (Experiments 1, 2, and 3) followed the guidelines described in Kitchenham et al. [47].

We presented this empirical study at the IEEE Conference on Software Testing, Regression 2014 [40]. The research question for this experiment is as follows:

RQ: Can our approach be effective in reducing the effort during regression testing?

The following sections describe the experiment details, which include objects of analysis, independent and dependent variables, measures, experiment setup and design, and threats to validity.

5.1.1. Objects of Analysis

In this experiment, we used five open-source PHP web applications with multiple versions.

OsCommerce [58] (open-source Commerce) is a web-based store-management and shopping cart application. This application has multiple add-ons developed by its large user community. *FAQForge* [56] enables users to create frequently asked questions (FAQs), documents, and manuals. *PhpScheduleIt* [71], is a web-based reservation and scheduling system. It provides a wide range of configurations: for example, administrators can set up access control that grants or denies access to reserve a resource for a user. *Mambo* [55] is a content management system for building various websites such as blogs, forums, and polls. It offers a robust set of API and caching mechanisms that enable building websites that are responsive even under high traffic conditions. *Mantis* [57] is a widely used web-based bug tracking system. It supports a wide array of browsers, operating systems, and mobile devices. *Mantis* integrates with source control tools such as *Subversion*, and with other bug-tracking systems such as *Bugzilla*. All these applications have many active users and have support groups that discuss the usage of existing features, issues, workarounds, and suggestions for improvement.

Table 5.1 lists, for each of the five applications, data about its associated "Version", "Lines of Code" and "Number of Files." Lines of code metric includes both PHP script and HTML.

Table 5.1. Objects of Analysis in Experiment 1.

Application	**Version**	**Lines of Code**	**Number of Files**
FAQForge	1.3.0	1806	20
	1.3.1	1837	20
	1.3.2	1671	18
osCommerce	2.2MS1	53510	302
	2.2MS2	68330	506
	2.2MS2-06017	78892	502
Mambo	4.5.5	149868	703
	4.5.6	150967	719
	4.6.1	127309	771
	4.6.2	129235	659
	4.6.3	130716	654
	4.6.4	133420	663
	4.6.5	133475	663
phpScheduleIt	1.0.0	35045	90
	1.1.0	59753	143
	1.2.0	63138	178
Mantis	1.1.6	139124	496
	1.1.7	139196	496
	1.1.8	139194	496
	1.2.0	206150	748
	1.2.1	206492	747
	1.2.2	207123	746
	1.2.3	209104	753
	1.2.4	209345	753

5.1.2. Variables and Measures

5.1.2.1. Independent Variables

The independent variable in this study is the test input generation technique. The experimental control (baseline) is the technique that generates executable test cases without utilizing reusable input constraint values. The heuristic technique generates executable test cases considering the reusability of input constraint values.

5.1.2.2. Dependent Variable and Measures

The dependent variable is the number of reusable input values identified using the proposed technique. We also measured the percentage of reusable input values over the total number of input values.

5.1.3. Experiment Setup

This experiment was performed on a standalone machine. The operating system on the machine was *Ubuntu Desktop 16.04*. To support the web-based applications, an *Apache* HTTP server was used. *MySQL* was installed to support the DBMS that is required for the object programs. PHP version 5.6, PHC 0.3 and PHPUnit 5.3 were installed to run and test PHP web-applications. Our tool was written in *Java*, and Perl and Bash scripts were used to automate various processes of the experiment. As described in Section 3.2.1, the process of identifying reusable constraints begins with *Preprocessing*. The Perl script takes the source code path to original and modified versions of the SUT, and generate ASTs. Then, impact analysis was performed by analyzing these two sets of ASTs, and finally, reusable constraint detection was performed using the impact analysis results.

We measured the total number of input values (*Total*) required for the new test paths and the number of reusable input values (*Reuse*) to calculate the percentage of reusable input values (*Reuse*/*Total*).

5.1.4. Data and Analysis

The results of this study and data analysis considering the research question are presented in this section. The research question of this study is to assess the effectiveness of the proposed approach in terms of the costs of generating new test cases for regression testing. The following two sets of data were collected to answer our research question:

1. The total number of input values required to execute new test paths
2. The number of reusable input values

These two sets of data collected from the experiment are summarized in Table 5.2. The table also includes the total number of paths and regression test paths. Traversing these paths enables us to compute the number of input test values required.

The total number of constraints across all version pairs of *FAQForge*, *Mambo*, *Mantis*, *osCommerce* and *phpScheduleIt* were 298, 15,863, 21,926, 8,462 and 3,376, respectively. The total number of constraints used were 83, 5,925, 9,887, 5,973 and 2,051, respectively for all version pairs of *FAQForge*, *Mambo*, *Mantis*, *osCommerce* and *phpScheduleIt*.

For each application, Table 5.2 shows "Version Pair", which is a pair of consecutive versions that were analyzed. "No. of Paths (*Pt*)" indicates the total number of paths in the newer version in the version pair. "No. of Regression Paths (*Pp*)" is the total number of paths generated by PARTE (the paths that exercise the modified code in the newer version). *Pp* is a subset of *Pt*. "No. of Input Values Required for *Pp* (*It*)" is the total number of input values required for the regression paths *Pp*. "No. of Reusable Input Values in *Pp* (*Ir*)" is the total number of input values that can be reused from the previous version for the regression paths. *Ir* is a subset of *It*. A visual representation of this data is shown in Figure 5.1, where the y-axis in the graphs is the percentage of input values that are reusable. The remainder of this section discusses the results of each application.

5.1.4.1. Results for FAQForge

As shown in Table 5.2, the version pair 1.3.0 and 1.3.1 has the total number of paths as 73, the number of regression paths as 5, the number of input values required to execute these paths as 25, and 19 of the 25 input values are reusable.

Manual code inspection of the changes introduced in version 1.3.1 shows that only one library file was affected. PARTE stores library files in a folder named '*lib*' and these files are not analyzed because they are not part of the application's source code. However, during the *Preprocessing* phase, if a PHP source code file of the application references a library using '*include*' and '*require*', then the library code is inserted into the PHP file. If there was any change in the library code, as in this version of *FAQForge*, *Path Generator* would recognize the change and create a test path corresponding to the change. Hence, this resulted in 5 regression paths for version 1.3.1.

The number of changes between versions 1.3.1 and 1.3.2 is higher compared to the first pair. Several source code changes across 3 files and a change to a library were observed during manual inspection. These modifications caused *Path Generator* to identify 19 regression paths. To execute these paths, 210 input values were needed. However, only 15 input values were determined to be reusable. The lower reuse is due to the source code changes that modified the def-use of variables between versions. Compared to the control technique, which generates tests and their input values, the results indicate that the proposed approach requires a relatively smaller number of test input values and some of which are reusable.

FAQForge is a smaller application than the other four considered in the experiment. The latest version of *FAQForge* has 18 files with around 1671 lines of code. Hence the number of paths and input values required is relatively small compared to other applications.

Table 5.2. Results of Reusable Constraint Values.

Application	**Version Pair**	**No. of Paths (*Pt*)**	**No. of Regression Paths (*Pp*)**	**No. of Input Values Required for *Pp* (*It*)**	**No. of Reusable Input Values in *Pp* (*Ir*)**
FAQForge	1.3.0 & 1.3.1	73	5	25	19
	1.3.1 & 1.3.2	73	19	210	15
Mambo	4.5.5 & 4.5.6	1357	65	490	294
	4.5.6 & 4.6.1	1388	1388	4446	67
	4.6.1 & 4.6.2	1416	236	2075	536
	4.6.2 & 4.6.3	1409	92	1236	250
	4.6.3 & 4.6.4	1444	114	1567	191
	4.6.4 & 4.6.5	1444	20	324	57
Mantis	1.1.6 & 1.1.7	3482	221	1524	708
	1.1.7 & 1.1.8	3482	185	1238	662
	1.1.8 & 1.2.0	4345	2802	20425	195
	1.2.0 & 1.2.1	4373	166	1772	558
	1.2.1 & 1.2.2	4389	106	1042	296
	1.2.2 & 1.2.3	4403	103	890	267
	1.2.3 & 1.2.4	4419	199	2643	263
osCommerce	2.2MS1 & 2.2MS2	2403	1719	4392	702
	2.2MS2 & 2.2MS2-060817	2409	58	813	219
phpScheduleIt	1.0.0 & 1.1.0	481	338	2389	280
	1.1.0 & 1.2.0	518	314	3877	472
	1.2.0 & 1.2.12	529	154	2339	861

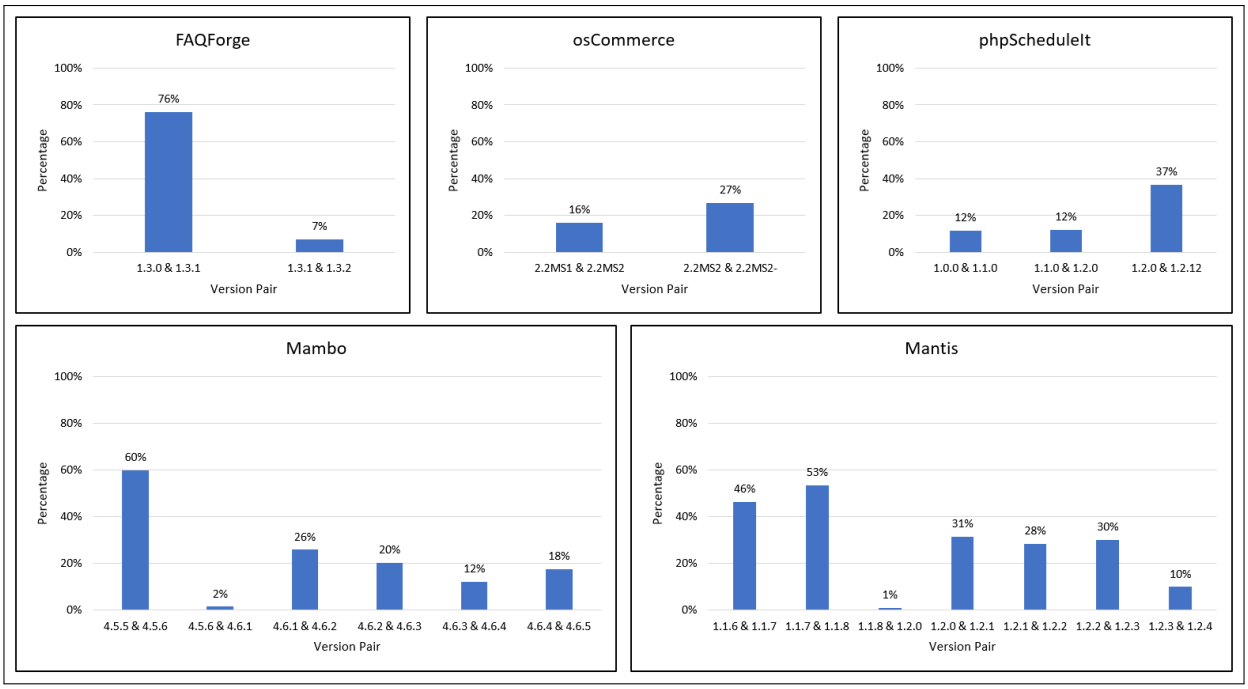

Figure 5.1. Input Value Reuse Rates.

5.1.4.2. Results for Mambo

Six version pairs of *Mambo* were analyzed. As shown in Table 5.2. Between versions 4.5.5 and 4.5.6 the number of functional changes were minor, and hence a higher reuse of input values was obtained. The release notes for version 4.6.1 mention that this was a major update, which included fixes and feature upgrades. The type and number of changes introduced in this version caused 1,388 regression paths with 4,446 input values. The def-use of several variables changed, which prevents reuse of many input values.

The remaining four version pairs had changes between the versions. Security patches to address cross-site scripting issues and minor bug fixes were introduced in these versions.

5.1.4.3. Results for Mantis

In *Mantis*, version pair 1.1.8 and 1.2.0 had lower reuse of input value. A manual inspection of source code changes and release notes for version 1.2.0 shows that many files were modified, several bug fixes were introduced, and feature were upgraded. These changes resulted in 2,802 regression test paths. These regression paths required 20,425 input values, with only 195 reusable input values.

Version pair 1.1.7 and 1.1.8 had a greater reuse since the number of source code changes between the versions was quite low.

Mantis is the largest application that was analyzed in the experiment. The latest version of Mantis has around 700 files and more than 200 KLOC. Given the size of *Mantis* reusing 1 in 4 input values save in the cost of solving constraints, and thus the proposed approach reduces the overall cost of regression testing.

5.1.4.4. Results for osCommerce

In *osCommerce*, the first version pair i.e., 2.2MS1 and 2.2MS2, the number of regression paths identified was 1,719, with 4,392 input values needed to execute these paths. Between these versions, a total of 279 files were modified. Although most changes were in the library files, there were a few modifications in each module of the application. Thus, only 702 input values were determined to be reusable.

In the second version pair, 2.2MS2 and 2.2MS2-060817, 105 of the 502 files had changes. The number of changes and files modified was lower compared to the first version pair. Fifty-eight regression paths, with 813 input values required to execute those paths, were identified. Given the type of changes, 219 input values were determined to be reusable.

5.1.4.5. Results for phpScheduleIt

A manual inspection of release notes and source code showed that new functionality and bug fixes were introduced in *phpScheduleIt* versions 1.1.0 and 1.2.0 while version 1.2.12 included a couple of minor patches. As observed with other applications in the experiment, when the number of changes is greater, the regression paths identified will also be greater in number, more input values will be needed to execute those paths, and input value reuse will decline. Version 1.1.0 introduced support for multiple administrators, web-configurable announcements, and localization features. Version 1.2.0 added features such as user groups, the ability to reserve additional resource for a given reservation, and support for user time zones. Introducing such major features caused a higher number of regression test paths and a reduced reuse. Version pair 1.2.0 and 1.2.12 had higher reuse due to the lower number of changes between those versions.

These results demonstrate that the proposed technique requires fewer input test values than the control technique and can reuse input values. To investigate whether the difference we observed is statistically significant, we performed the paired t-test at a significance level of 0.05. The test results indicate that there is a significant difference between techniques (p-value < 0.01), meaning that our approach was significantly effective in reducing input test values.

5.1.5. Threats to Validity

In this section, the internal and external threats to the validity of this study are explained. The approaches used to limit the impact of these threats are also described.

5.1.5.1. Internal Validity

The outcome of this study could be affected by the choice of program analysis. We applied static analysis on a dynamic programming language, PHP. To do so, we had to change many statements in the program during the file preparation phase, removing and fixing many dynamic environment variables. However, the process was carefully examined to minimize the adverse effect that might be introduced into the modified files.

5.1.5.2. External Validity

This study used open-source web applications, so these programs are not representative of the applications used in practice. Therefore, the results cannot be generalized. However, we tried to reduce this threat by using five non-trivial sized web applications with multiple versions that have been utilized by many users.

5.1.6. Discussions

Our experiment results strongly suggest that the approach that supports input variable value reuse for regression tests can save a substantial amount of effort during regression testing.

As we observed from the data analysis, overall many version pairs of the applications produced the high reuse rates of input values with our approach over the control technique (original PARTE approach). For some version pairs, the reuse rates were exceptionally high. The first version pair (1.3.0 and 1.3.1) for *FAQForge* yielded a 76% reuse rate, and the first version pair (4.5.5 and 4.5.6) for *Mambo* yielded 60%. Further, two version pairs for *Mantis* (1.1.6 and 1.1.7, and 1.1.7 and 1.1.8) showed reuse rates of 46% and 53%, respectively.

Because the test paths require actual inputs to create executable test cases, incorporating this approach to the PARTE approach that we introduced in our previous work can reduce regression testing effort substantially. For instance, in the case of *osCommerce*, the number of test paths generated using program slice information was 1,719 for version 2.2MS2. For those test paths, we learned from our previous study [59] that a large number of constraints were not solvable by automatic resolvers. Instead, many input values had to be resolved manually, and it required a lot of effort. Our approach that facilitates constraint reuse allowed a substantial reduction of efforts during the constraint resolution process.

We also observed that there is a relationship between the reuse rate and the number of changes in the version pair. For *FAQForge*, our approach identified more reusable input values from the first version pair

than the second version pair. Our manual inspection of the source files for the versions revealed that there are fewer changes in the first version pair than in the second version pair. Similarly, for *osCommerce*, more reusable input values were identified in the second version pair than the first version pair. Our inspection revealed that, for the second version pair, the number of changed files was fewer than the first version pair.

In addition to the impact of the amount of changes to the reuse rate, we also observed that the type of changes between versions can impact the reuse rate.

In *phpSchedulelt*, the changes in the last version pair were related to language support and several minor bug fixes, whereas the changes in other version pairs involve new feature additions and several bug fixes. For example, version 1.1.0 introduced a feature for multiple day reservations and version 1.2.0 allowed additional resources to be added to an existing reservation. To support these new features, several new classes were added in those versions, and thus their input reuse rates were relatively low compared to the last version pair.

In the case of *Mambo*, the second version pair went through significant code changes including an external editor integration into *Mambo* and a security feature addition. The existing code had to be modified to support these new features, and as a consequence, the majority of input constraints generated for the previous versions could not be reused (the reuse rate was 2%). But, the first version pair involved minor bug fixes, so the reuse rate was relatively high (60%).

In the case of *Mantis*, for the third version pair (1.1.8 and 1.2.0), 149 files had changed. The changes involved bug fixes and several new feature additions, such as monitoring bugs. Similar to *Mambo*, these new additions affected the existing code, thus the reuse rate (1%) was exceptionally low. The version pair, 1.1.6 and 1.1.7, involved only 18 minor bug fixes, and no new source code or resource files were added. Thus, the reuse rate (46%) was relatively high compared to other version pairs.

These observations clearly state the fact that application versions with fewer changes or changes introduced in small patches are more beneficial in reducing test case generation costs than versions with a large number of changes. In general, irrespective of the number of changes in program source code, reusing constraints reduced the new test case generation effort by a substantial amount.

To our knowledge, this study was the first attempt to investigate the reusability of variable constraints and their input values. The proposed approach produced promising results and the findings from the study provided an insight about how reusable constraint values can be utilized during the testing and regression testing process.

Encouraged by the results of this experiment we pursued our research on how existing unit tests can be utilized effectively, and avoid the costs of generating new tests so that the overall cost of regression testing can be further reduced. In our next chapter we describe our second experiment that investigates on how reusable tests can be identified using constrain reuse information.

5.2. Experiment 2: Test Selection with Reusable Constraint Values

The results of Experiment 1 suggest that the proposed approach that supports input variable value reuse for regression tests can save a substantial amount of effort during regression testing. In our second experiment, we explore additional cost savings by considering test selection that utilizes existing test cases and reusable constraint values when applications possess existing test cases. We submitted this study to Elsevier's Information and Software Technology [26]. Our second experiment addresses the following research questions:

RQ1: Is our test selection approach that selects existing test cases and reuses constraint values effective in reducing the overall effort for testing new versions of programs?

RQ2: Do the selected tests have the same fault detection ability as the original tests?

The following subsections describe the objects of analysis, experiment setup, threats to validity, and interpretation of the results.

5.2.1. Objects of Analysis

This experiment requires existing test cases; thus, we utilized a subset of the object programs and versions used in Experiment 1 that have test cases, including one additional application (*log4php*). *Log4php* [53] is a logging framework for PHP. It is configurable through xml, properties or PHP files and it has built-in logging formats that include HTML, XML or any user defined pattern. It has been used in many large open-source applications, such as *vtiger*, *scalr* and *Helpzilla*, and commercial projects such as *sitesupra*. See Section 5.1 for descriptions of remaining applications: (*Mantis*, *Mambo*, and *phpScheduleIt*). We chose these specific versions of the applications in this study because the changes between the versions addressed customer reported bugs and included fixes for security issues.

Test cases for *Mantis* and *log4php* came with these programs. Although these tests cover the core functionality of the applications, the overall coverage was quite low. A few modifications between the versions were not covered by any of the existing tests. Hence, we manually created new tests, verified that

the new tests cover the missing gaps in the core functionality and thus, increased the code coverage. *Mambo* and *phpScheduleIt* do not have any existing tests, so we created test cases for them. When we create these test cases, we focused on covering the core functions of each application, so the number of test cases is rather small compared to two other applications.

Table 5.3 shows the object programs and their associated data. "Number of Existing Tests" shows the number of existing test cases for the corresponding version. "Coverage of Existing Tests" is the code coverage of existing tests. "Number of New Tests" is the number of new tests that we created. "Total Number of Tests" is the sum of "Number of Existing Tests" and "Number of New Tests". "Total Coverage of Tests" is the code coverage corresponding to "Total Number of Tests".

Table 5.3. Objects of Analysis in Experiment 2.

Application	**Version**	**Lines of Code**	**Number of Files**	**Number of Existing Tests**	**Coverage of Existing Tests**	**Number of New Tests**	**Total Number of Tests**	**Total Coverage of Tests**
Mantis	1.2.0	206150	748	62	41%	32	94	48%
	1.2.1	206492	747	64	41%	33	97	49%
	1.2.2	207123	746	66	42%	42	108	51%
	1.2.3	209124	753	70	44%	54	124	53%
	1.2.4	209345	753	71	44%	59	130	57%
Mambo	4.5.5	149868	703	0	0%	65	65	36%
	4.5.6	150967	719	0	0%	76	76	39%
	4.6.1	127309	771	0	0%	83	83	42%
	4.6.2	129235	659	0	0%	90	90	44%
	4.6.3	130716	654	0	0%	102	102	53%
	4.6.4	133420	663	0	0%	105	105	54%
	4.6.5	133420	663	0	0%	110	110	58%
phpScheduleIt	1.0.0	35045	90	0	0%	40	40	16%
	1.1.0	59753	143	0	0%	45	45	17%
	1.2.0	63138	178	0	0%	52	52	20%
	1.2.12	72396	192	0	0%	55	55	22%
log4php	2.0.0	12014	97	39	37%	23	62	59%
	2.1.0	13702	105	44	39%	31	75	62%
	2.2.0	15132	114	46	40%	50	96	73%
	2.3.0	16720	132	49	42%	49	98	74%

5.2.2. Variables and Measures

5.2.2.1. Independent Variables

The independent variable in this study is the regression test selection technique. The experimental control is the technique that uses all existing test cases when testing the modified version of the program.

5.2.2.2. Dependent Variable and Measures

The dependent variables in this study are as follows:

1. The number of reusable tests selected for the modified version of the program.

2. Mutation score for tests considered in the selection technique.

5.2.3. Experiment Setup

This study was performed on a standalone machine. The operating system on the machine was *Ubuntu Desktop 16.04*. To support the web-based applications, an *Apache* HTTP server was installed and configured. *MySQL* was installed to support the DBMS needs of the object programs. PHP version 5.6, PHC version 0.3, and PHPUnit 5.3 were installed to run and test PHP web applications. PARTE tools are written in *Java*. For this experiment, PARTE tools were built and executed with *Java* 8 Update 91 version. To control data flow between tools, Perl and Bash scripts were used. As described in Section 5.1.3, we used a Perl script to generate ASTs, PDGs, slices and to identify reusable constraints. Next, the Perl script is invoked to perform test selection by passing the paths to the root folder containing the impact analysis results of the original and modified version of SUT and the path to the folder containing the tests for the original version of SUT.

We reviewed of the changelog and manually inspected the code to verify if there are real defects between the version pairs of the applications. Although some version pairs have such defects, many of them did not. Hence, as a substitute [42], we chose mutation analysis to investigate whether the selected tests catch the same faults as the original tests, which shows the selected tests maintain the same fault detection ability.

To create mutants, we used *Humbug* [41], which is a mutation testing framework for PHP. *Humbug* generates a basic set of mutants such as changing binary arithmetic operators, boolean values, conditional boundaries, increments, and return values. *Humbug* selects existing tests that cover the statement, executes the selected tests on mutated code, and verifies if the mutant is killed. *Humbug* uses PHPUnit to execute the

tests, and it produces the mutation score for each set of test cases as the final outcome. Since our focus is regression testing, we use only generated mutants that fall within modified areas of code.

5.2.4. Data and Analysis

This section presents the results of the second experiment and an analysis of the data.

5.2.4.1. RQ1 Analysis

RQ1 investigates if the proposed regression test selection approach is effective in identifying tests that cover modifications between the version pairs, and if augmenting this selection with constraint value reuse information reduces the overall costs of regression testing.

Table 5.4 summarizes the results of the test selection. For each application, the table shows "Version Pair," which is a pair of consecutive versions of the application being analyzed. The first version in each pair is considered the old version, and the second is the modified version. The "Total Tests (*T*)" column shows the total number of unit tests available for the corresponding old version. "Selected Tests (*Ts*)" shows the total number of selected unit tests that should be run to regression test the modified version. Some selected test cases may be obsolete if their input values are no longer applicable to the modified version. "Obsolete Tests (*To*)" shows the number of obsolete tests. The "Final Tests (*Tf = Ts - To*)" column indicates the number of tests obtained after subtracting the obsolete tests (*To*) from the selected tests (*Ts*).

From this table, we observed that the number of unit tests tends to grow across versions for all applications because new tests are added to cover new features or improve existing coverage. Overall, in each version pair of all applications there were existing tests that covered the affected code paths. We discuss the results of each application in detail in the following subsections. The visual representation of rate of savings, which is the percentage of "Selected Tests" in "Total Tests", achieved in each version pair of an application is presented as a line graph in Figure 5.2.

We calculated the percentage change in the total execution time of "Total Tests" and "Selected Tests" for each version pair across all four applications. The highest and lowest percentage change in execution time observed was 73% and 0%, respectively (37% on average). The lowest percentage change was 0% since in *phpScheduleIt* version pair 1.0.0 & 1.1.0, all the existing tests were selected as regression tests. For all other version pairs across applications, the percentage change was more than 10%. This shows that the proposed test selection approach is effective in reducing the overall cost of regression testing.

Table 5.4. Results of Test Selection.

Application	**Version Pair**	**Total Tests (*T*)**	**Selected Tests (*Ts*)**	**Obsolete Tests (*To*)**	**Final Tests (*Tf = Ts - To*)**
Mantis	1.2.0 & 1.2.1	94	42	12	30
	1.2.1 & 1.2.2	97	33	11	22
	1.2.2 & 1.2.3	108	52	28	24
	1.2.3 & 1.2.4	124	67	59	8
Mambo	4.5.5 & 4.5.6	65	28	9	19
	4.5.6 & 4.6.1	76	27	18	9
	4.6.1 & 4.6.2	83	45	24	21
	4.6.2 & 4.6.3	90	47	25	22
	4.6.3 & 4.6.4	102	23	17	6
	4.6.4 & 4.6.5	105	82	72	10
phpScheduleIt	1.0.0 & 1.1.0	40	40	32	8
	1.1.0 & 1.2.0	45	32	21	11
	1.2.0 & 1.2.12	52	34	18	16
log4php	2.0.0 & 2.1.0	62	40	29	11
	2.1.0 & 2.2.0	75	52	31	21
	2.2.0 & 2.3.0	96	58	32	26

Table 5.5. Results of Mutation Testing.

Application	**Version Pair**	**Total Tests (*T*)**	**Selected Tests (*Ts*)**	**Total Mutants (*M*)**	**Mutation Score for *T***	**Mutation Score for *Ts***
Mantis	1.2.0 & 1.2.1	94	42	573	100	100
	1.2.1 & 1.2.2	97	33	604	100	100
	1.2.2 & 1.2.3	108	52	609	100	100
	1.2.3 & 1.2.4	124	67	763	100	100
Mambo	4.5.5 & 4.5.6	65	28	560	100	100
	4.5.6 & 4.6.1	76	27	622	100	100
	4.6.1 & 4.6.2	83	45	740	100	100
	4.6.2 & 4.6.3	90	47	745	100	100
	4.6.3 & 4.6.4	102	23	785	100	100
	4.6.4 & 4.6.5	105	82	793	100	100
phpScheduleIt	1.0.0 & 1.1.0	40	40	367	100	100
	1.1.0 & 1.2.0	45	32	532	100	100
	1.2.0 & 1.2.12	52	34	758	100	100
log4php	2.0.0 & 2.1.0	62	40	339	100	100
	2.1.0 & 2.2.0	75	52	415	100	100
	2.2.0 & 2.3.0	96	58	435	100	100

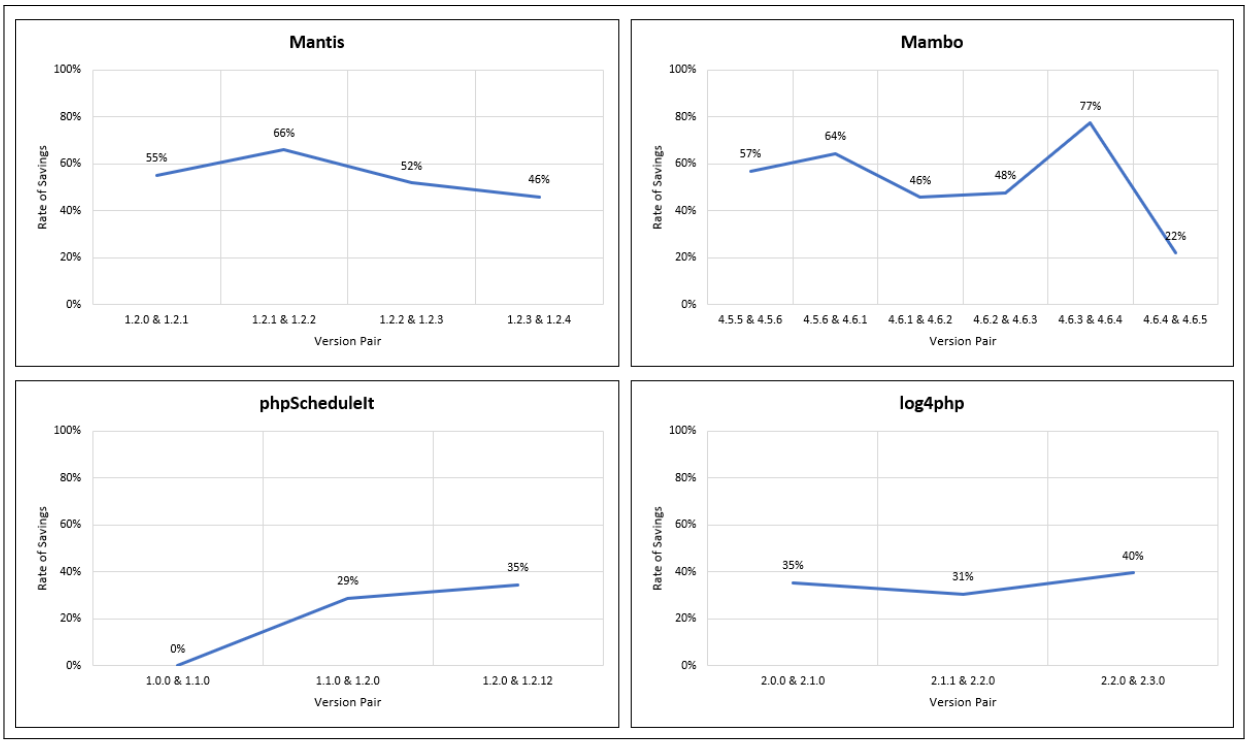

Figure 5.2. Rate of Savings in Test Selection.

5.2.4.2. Results for Mantis

As shown in Table 5.4, for some version pairs, the portion of obsolete tests is somewhat high. Version pair 1.2.3 and 1.2.4 has the highest number of obsolete tests (59), which means that only 8 tests can be reused to test version 1.2.4.

Upon manual inspection of the source for version 1.2.4, we found that this version includes more than 30 fixes in different areas such as API changes, security problem fixes, and enhancements to user administration features. A few existing tests (67) of version 1.2.3 covered the code paths that had undergone modification in version 1.2.4. An example of such modification is in '*account_update.php*' that introduced the '*t_ldap*' variable to check if the current session is using LDAP. Tests that cover the paths where '*t_ldap*' was introduced, for example a test that invokes '*account_update.php*' through a call from '*account_update.php*', will behave differently between 1.2.3 and 1.2.4. Modifications of this kind caused the def-use of a few variables to change between the versions. Hence, tests that had input values for such variables, a total of 59 tests, became obsolete, leaving 6 existing tests that can be reused for regression testing 1.2.4. Although new tests were added to version 1.2.4, only 2 covered enhancements in user administration features. Thus, a total of 8 tests made up the final set of selected tests.

In contrast, version pair 1.2.0 and 1.2.1 had fewer functional changes. Existing tests in version 1.2.0 did not have coverage for localization and webpage components that were modified in 1.2.1. The tests selected (42) for 1.2.1 covered the security and API changes that did not change the def-use of many variables in the affected paths. Thus, only 12 of the tests were obsolete leaving 30 in the final set of selected regression tests that can be reused for 1.2.1.

Thus, it can be observed that the rate of savings is affected by factors such as the type of code changes and the coverage of existing tests. Utilizing constraint reuse information enables test teams to filter obsolete tests automatically, and thus identify a final set of regression tests that can be reused with existing values.

5.2.4.3. Results for Mambo

On reviewing the source code for *Mambo* version 4.6.1, we found that this version was a feature update that introduced new features and improved many existing ones. Between these two releases, there were two release candidates of version 4.6. This caused substantial code churn to the point where 40% of the source code files were modified between versions 4.5.6 and 4.6.1, resulting in 27 of the 76 tests being selected. Given the type of code changes, the def-use of a few variables did not remain the same between versions, causing 18 tests to become obsolete. Thus, the final set of selected tests included 9 tests.

A manual inspection of *Mambo* version 4.6.4 shows that this version introduced new security features such as hardened security in '*/includes/Cache/Lite/Output.php*', included fixes for several bugs, and refactored code. These types of functional changes introduced new code paths, removed code paths that had existed in earlier versions, and modified the def-use of common variables between the versions. *Mambo* version 4.6.3 had 102 tests. PARTE's test selection identified 23 tests that cover the affected paths. Given the def-use changes, there were 17 obsolete tests, leaving only 6 in the final set of selected regression tests.

Version 4.6.5 had minor functional changes. Only 2% of the code base had undergone modifications, and many of those were in non-executable code. However, the few functional changes that were made in this version were in the code paths that influence common usage scenarios in *Mambo*. For example, a definition of the variable '*$link*' was removed from the '*saveContent*' and '*cancelContent*' functions in '*components/com_contentcontent.php*'. These types of changes caused 82 tests to be selected as regression tests. The def-use of variables that were used in multiple existing tests was modified in the new version. Hence, 72 of the 82 selected tests were obsolete, resulting in a final set of 10 regression tests that can be reused with existing values to test version 4.6.5.

Like *Mantis*, number of selected tests in *Mambo* was dependent not only on the number of code changes but also on the component that was modified. As the number of tests increases, effectively selecting tests and automatically detecting obsolete tests will minimize the manual verification needed to validate test inputs. Thus, the proposed approach reduces the overall cost of regression testing.

5.2.4.4. Results for phpScheduleIt

phpScheduleIt is smaller than *Mantis* and *Mambo*. A manual inspection of version 1.1.0 showed that some core components of the application had undergone modification. For example, in version 1.0.0, a function that deletes a reservation does not verify if the current user has sufficient privileges to delete a reservation [72]. This was fixed in 1.1.0, in which a condition was added to check if the current user has sufficient access to delete a reservation. These types of changes affect the common use–case scenarios of an application. Thus, all 40 tests in version 1.0.0 were selected as regression tests for 1.1.0. Code changes modified the def-use of certain variables that 29 of the 40 tests depended on, causing these 29 tests to be marked as obsolete, which left 8 tests in the final set of regression tests that can be reused for version 1.1.0.

In contrast, the changelog for version 1.2.12 showed fewer functional changes than the other two version pairs. Hence, 34 of 52 tests were selected. Given that the def-use of a few variables changed, 18 tests were identified as obsolete, and the remaining 16 tests were in the final set of selected regression tests.

5.2.4.5. Results for log4php

log4php is the smallest of all the applications in this experiment, with around 12 KLOC. A review of source code and changelog for version 2.1.0 reveals that several functional changes were made in the '*LogConfigurator*' and '*LogAppender*' modules. For version 2.0.0, it was reported [54] that connections to the database were not closed correctly. A malicious user could trick the application into leaving multiple open connections, thus making the database inaccessible to other users. A fix for this issue was made in the destructor of '*LoggerAppenderPDO*' class in version 2.1.0. Fixes in such critical areas caused 40 of the existing 62 tests in version 2.0.0 to be selected as regression tests for 2.1.0. But the types of changes modified the def-use of a few variables, causing 29 tests to become obsolete. Note that there were several nonfunctional changes that had no effect on the number of tests selected. Thus, 11 tests finally remained in the set of selected tests that can be reused to regression test version 2.1.0. A similar inspection of version pair 2.2.0 and 2.3.0 showed relatively fewer functional changes. Hence, the number of obsolete tests was lower, and the final count of regression tests selected was slightly higher for this pair.

To investigate whether the difference we observed from the results of these applications is statistically significant, we performed the paired t-test at a significance level of 0.05. The test results indicate that there is a significant difference between techniques (p-value < 0.05), meaning that our approach was significantly effective in reducing the number of test cases to be rerun.

5.2.4.6. RQ2 Analysis

RQ2 investigates whether the selected test cases have the same fault detection ability as the entire test cases.

Table 5.5 summarizes the results of mutation testing experiment. In Table 5.5, "Number of Mutants (M)" is the number of mutants generated for the newer version in the version pair. "Mutation Score with T" is the mutation score obtained when all tests were executed against *M*. "Mutation Score with Ts" is the mutation score obtained when selected tests *Ts* were executed against *M*.

From Table 5.5, we can observe that the fault detection ability of the selected tests is same as the original test cases across all versions and applications. From this result, we can say that by selecting a subset of test cases that is required for testing the modified version of the program, we were able to save time and effort while providing the same fault detection ability as the entire test cases.

5.2.5. Threats to Validity

This section describes the internal and external threats to validity of this experiment. Also describes the techniques that reduce the effect of these threats.

5.2.5.1. Internal Validity

Our experiments depend on PARTE tools; thus, any defects in the tools can affect experiment results. To control this threat, we validated our tools on several simple PHP programs, and we thoroughly examined the tools and removed existing inconsistencies. Partial matching for statement values of a variable sometimes becomes tricky for complex expressions. However, we examined different scenarios to avoid any discrepancy with partial matching.

5.2.5.2. External Validity

Three issues limit the generalization of our results. The first issue is application representativeness. Our experiments used open-source web applications, so the results we obtained might not be applicable to industrial applications and other types of web applications. However, to reduce this threat, we used large, actively used PHP web applications with multiple versions, and active support groups, and this threat can be further addressed through additional experiments with wider populations of applications. The second issue

is test case representativeness. We used unit tests in our experiment, but different types of test cases can produce different results. While the use of a single type of test cases can be a threat to validity, unit tests have been actively used by both open-source community (most of the applications we used came with unit tests) and industry (e.g., I can confirm that the majority of Microsoft's products have been tested using unit tests), thus we believe that the type of test cases used in our experiment can be adopted in the industrial practice. Nevertheless, to address this threat, further experiments with different types of test cases should be performed. The third issue involves fault representativeness. We used mutation faults in our experiment, and although there is some evidence that mutation faults can be representative of real faults for purposes of experimental evaluation [4], the distribution of mutation faults used in our experiment may not match distributions of faults found in practice. The additional experiments with different types of faults (e.g., real faults) should be conducted to address this threat.

5.2.6. Discussions

Results from this experiment clearly demonstrated that our approach is effective not only in selecting existing tests that cover the regression paths but also identifies reusable and obsolete tests. Thus, the manual effort that goes into activities such as the selection of tests and ensuring that the input values of the selected tests are applicable in the modified version, is eliminated or greatly minimized.

Table 5.4 shows that there are existing tests that can be reused to regression test the modified version of an application. A few of the selected tests have reusable inputs, meaning the test covers the modified code paths and no changes are needed to the input values of those tests. Also, it can be observed from the results that each version pair resulted in a few obsolete tests. As can be inferred from Table 5.4, the percentage of selected tests varied between 23% and 100%, the percentage of reusable tests ranged from 6% to 32%. and the highest and lowest percentage of obsolete tests was 80% and 11%, respectively. Percentages were computed relative to the total number of tests. Table 5.5 demonstrates the effectiveness of our test selection approach. The ability to find a fault i.e., mutation score of the selected tests is same as that of all the tests. This means the selected tests set is as effective as the entire test suite.

We learned that the number of tests selected for regression testing depends on three factors as listed below.

1. The coverage of existing tests.

2. The type or the location in the source code where the change happened.

3. The number of changes between the original and modified version.

4. Rate of reusable input values.

For example, consider the change between *Mantis* version 1.2.3 and 1.2.4. A simple change was introduced in a core API, '*account_update.php*'. This path was covered by lot of existing tests. These tests were selected as regression tests. However, the API change impacted the def-use of lot of variables. Due to the changes to def-use the selected tests were not reusable. In contrast, there were several changes between versions 1.2.0 and 1.2.1. However, these changes were made to support localization. Existing tests did not cover localization and hence no test was selected. Changes unrelated to localization were covered by a few existing tests, and those were selected as can be observed in Table 5.4.

In *Mambo*, the were significant (type and quantity) changes between versions 4.5.6 and 4.6.1. This caused a fewer, compared to other version pairs, tests to be selected, and a smaller number of reusable tests. In *phpScheduleIt*, a security fix was introduced in version 1.1.0. The most common use-case scenarios of *phpScheduleIt* were affected by this fix. Though there were tests that cover the modified code path, the changes to def-use of the variables made the tests obsolete. In *log4php*, the effect of the three factors i.e., test coverage, type and quantity of changes, can be observed across all version pairs.

From these observations, it is clear that there are existing tests that can be reused to regression test the modified version of an application. The cost of generating new tests and input values can be avoided.

Review of literature showed us that this experiment is the first attempt to augment regression test selection with reusable input values for regression testing of web applications. The results from the proposed approach provided insights about how reusable input values can be utilized to improve the effectiveness of regression test selection, and reduce the overall costs of regression testing of web applications.

Results of Experiment 1 and 2 greatly motivated us to further explore the benefits of reusability of existing resources. Obsolete tests identified in Experiment 2 can be replaced by generating new tests. However, generating a new test to replace the original test, is costly and the intent of the original test will not be retained. Hence, we pursued our research on repairing the obsolete tests using two different test repair approaches. In the next section we describe our experiment on evaluating our proposed test repair approach, results of the experiment and the insights gained from analyzing those results.

5.3. Experiment 3: Test Repair

We conducted an empirical study to assess the proposed approach of repairing obsolete tests. We submitted this study to Elsevier's Information and Software Technology [26]. The research question for this experiment is as follows:

RQ: What percentage of obsolete tests can be repaired using our approach?

The experimental details including the objects of analysis, independent and dependent variables, measures, design, setup, and the threats to validity are described in the following sections.

5.3.1. Objects of Analysis

In this experiment, we used five open-source PHP web applications that have multiple consecutive versions and tests. Four of these five applications are the same as those used in Experiment 2. Details of those four applications are shown in Table 5.3. A new application called *Composer* [15] was chosen for this experiment. *Composer* is a dependency manager that enables developers to declare, manage, and install dependencies for PHP web applications. The consecutive versions of *Composer* chosen for the experiment have a large number of existing tests, changes between the versions addressed customer reported bugs and included fixes for security issues. A few modified functions from the previous version were not covered by the existing tests, so we manually created a few tests to fill such gaps. Details of *Composer* are shown in Table 5.6.

Table 5.6. Additional Object of Analysis for Experiment 3.

Application	**Version**	**Lines of Code**	**Number of Files**	**Number of Existing Tests**	**Coverage of Existing Tests**	**Number of New Tests**	**Total Number of Tests**	**Total Coverage of Tests**
Composer	1.0.0	42792	398	562	73%	19	581	76%
	1.1.0	44039	409	570	71%	8	578	78%
	1.2.0	45406	419	592	72%	11	603	76%
	1.3.0	46783	429	624	68%	27	651	73%
	1.4.0	47408	431	627	69%	4	631	73%
	1.5.0	48290	434	652	67%	7	659	71%

5.3.2. Measures

To evaluate our approach, we measured the total number of tests that were repaired (*Repaired*), the total number of obsolete tests (*Obsolete*), and the percentage of repairable tests (*Repaired*/*Obsolete*).

5.3.3. Experiment Setup

This study was performed on a desktop machine running *Ubuntu* 16.04 operating system. PHP version 5.6, PHC 0.3 and PHPUnit 5.3 was installed on the machine. *Java* was required on the machine since PARTE and POTR are written in *Java*. To run the web-based applications, *Apache HTTP Server* was installed and configured. For the application's database requirement, *MySQL* was installed on the machine. As described in Section 5.2.3, a Perl script, is used to generate ASTs, PDGs, slices, identify reusable constraints, identify reusable and obsolete tests. To perform test repair through *Test Repair*, the Perl script takes the paths to the folders containing the obsolete tests, the tests for the original version and impact analysis results.

5.3.4. Data and Analysis

In this section, we present the experimental results and the data analysis considering our research question. Table 5.7 summarizes the results from our experiment. In this table, the column header 'Application' refers to the name of the application under test. 'Version Pair' is the two consecutive versions of the application, 'Unit Tests (*Tt*)' is the total number of unit tests available in the first version of the version pair. 'Obsolete Tests (*To*)' is the total number of obsolete tests identified from unit tests (*Tt*) (tests that are not applicable to the second version of the version pair). The obsolete tests were identified through the *Reusable Tests* component. 'Repaired Tests using MSA (*Tm*)' and 'Repaired Test using AA (*Ta*)' is the total number of obsolete tests repaired using Method Signature Analyzer and Assertion Analyzer, respectively. Finally, the last column, 'Percentage of Tests Repaired (*(Tm + Ta)/To*)' is the percentage of obsolete tests repaired using POTR. A visual representation of the results is shown in Figure 5.3.

From the results, we observed that all version pairs for all five applications contained obsolete tests. The percentage of obsolete tests repaired using our approach ranged between 25% to 100% with an average of around 70%. In particular, *Composer* produced relatively high repair rates across all version pairs (the lowest and highest was 63% and 100%, respectively). The repair rate was also high in *Mambo* where the average was 75%. One version pair in *Mambo* had a low repair rate due to the type of code changes that happened between two versions. In *Mantis* and *phpScheduleIt*, the average rate of repair was around 70%. In the case of *log4php*, the type of changes between the versions caused the repair rate to be the lowest among the five applications. MSA could not find a suitable repair for changes such as addition of a new

parameter without default values and modification of a parameter's data type in a method's signature. Hence, in *log4php* the average percentage of obsolete tests repaired was 38%.

The number of tests repaired using MSA is lower than AA, and this is partly due to the type of changes that happened in the new versions. The changes to a method signature often involved introduction of a new parameter where the new parameter was provided a default value. This type of change does not fail an existing test. Whereas, a method signature change that involved removal of a parameter, modification of the data type or ordering of a parameter, introduction of a new parameter with no default value will cause test failures. We learned that such changes are few and are considered breaking [98] since the established contract of the method no longer holds in the new version. Code changes within a method body are more common and such changes can cause assertions in tests to fail, and such failures are repaired by AA. AA had 100% success rate of fixing a failing assertion in an obsolete test. Whereas, the success rate MSA was low due to the various challenges as will be described with the examples in the following paragraph.

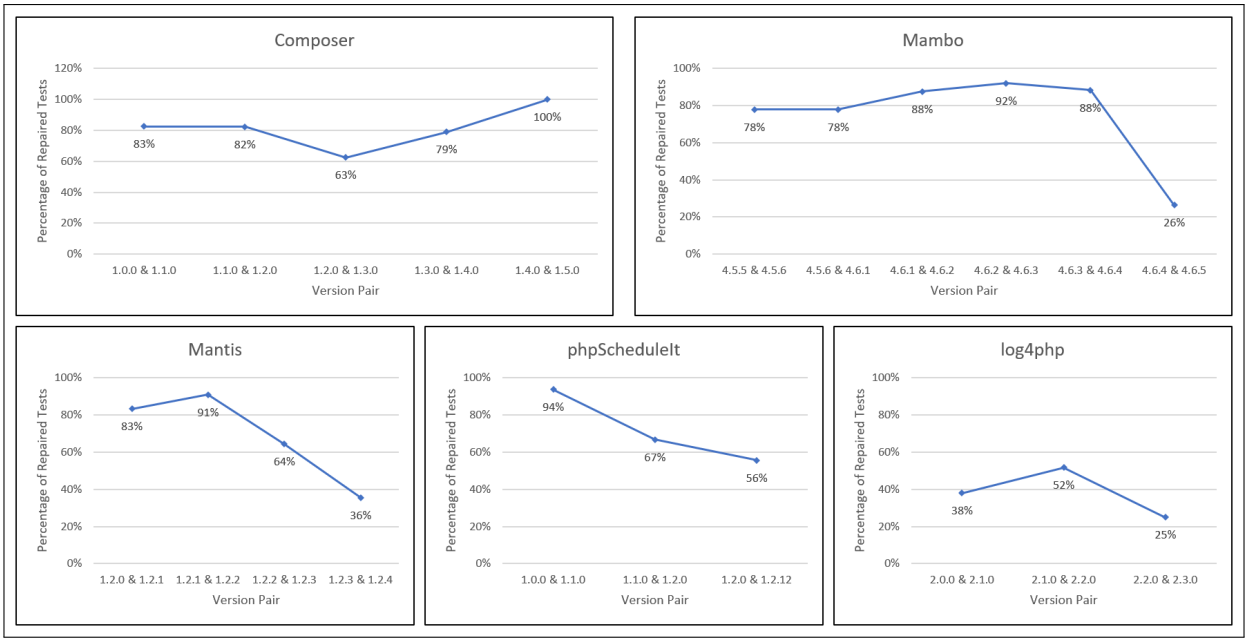

Figure 5.3. Percentage of Obsolete Tests Repaired.

In *Composer* version 1.1.0, a fix was introduced to ignore '*–no-dev*' option [18]. This fix removed '*withDevReqs*', a parameter from '*processDevPackages*' method. Two tests that invoked '*processDevPackages*' passing '*true*' as the parameter value for '*withDevReqs*', failed. MSA detected the change and repaired the test by removing '*withDevReqs*' parameter from the method invocation. In version 1.4.0 of *Composer*,

Table 5.7. Results of Test Repair.

Application	**Version Pair**	**Unit Tests (*Tt*)**	**Obsolete Tests (*To*)**	**Repaired Tests using MSA (*Tm*)**	**Repaired Tests using AA (*Ta*)**	**Percentage of Tests Repaired ((*Tm* + *Ta*)/*To*)**
Composer	1.0.0 & 1.1.0	581	23	6	13	83
	1.1.0 & 1.2.0	578	17	3	11	82
	1.2.0 & 1.3.0	603	8	2	3	62
	1.3.0 & 1.4.0	651	19	8	7	79
	1.4.0 & 1.5.0	631	7	2	5	100
Mantis	1.2.0 & 1.2.1	94	12	1	9	83
	1.2.1 & 1.2.2	97	11	4	6	91
	1.2.2 & 1.2.3	108	28	7	11	64
	1.2.3 & 1.2.4	124	59	6	15	35
Mambo	4.5.5 & 4.5.6	65	9	0	7	78
	4.5.6 & 4.6.1	76	18	1	13	78
	4.6.1 & 4.6.2	83	24	0	21	87
	4.6.2 & 4.6.3	90	25	1	22	92
	4.6.3 & 4.6.4	102	17	1	14	88
	4.6.4 & 4.6.5	105	72	3	16	26
phpScheduleIt	1.0.0 & 1.1.0	40	32	7	23	94
	1.1.0 & 1.2.0	45	21	2	12	67
	1.2.0 & 1.2.12	52	18	2	8	56
log4php	2.0.0 & 2.1.0	62	29	2	9	38
	2.1.0 & 2.2.0	75	31	3	13	52
	2.2.0 & 2.3.0	96	32	5	3	25

a parameter named '*ignoreFilters*' was added in '*archive*' [19] method causing an existing test to fail. Since the type of the new parameter was not specified and MSA could not determine a value that should be passed to this parameter, MSA failed to repair the test. Similarly, MSA was unable to fix existing tests that failed due to the changes in *Mambo* version 4.5.6. In this version, security fixes were performed in '*mod_login*' and '*mod_template_chooser*'. The fixes required adding new parameters to '*getSelectedValue*' method in '*mod_templatechooser.php*' file. A change of this kind is considered a breaking change since the change breaks the backward compatibility [98].

In all other instances where a parameter was added, and a default value was assigned to the parameter, MSA was able to correctly identify the new parameter and repair the test by including the new parameter in the method invocation. A test can fail if the modified version of the method signature specifies the type on a parameter. For example, between versions 2.1.0 and 2.2.0 of *log4php*, the method '*log($priority, $message, $caller = null)*' was changed to '*log(LoggerLevel $level, $message, $throwable = null)*'. Observe

that the first parameter, was renamed to '*$level*' and the type specified as '*LoggerLevel*' in the new version. Although the method body was not modified, a test in 2.1.0 failed since the input provided for the first parameter was not strictly of type '*LoggerLevel*'. MSA repaired the test by passing an object of type '*LoggerLevel*' created in the test setup method. A modification to the order of parameters in a method signature is uncommon, but it did occur between version 1.2.3 and 1.2.4 of *Mantis*. MSA repaired the test by identifying the order of parameters that allowed the test to pass. Providing such repairs will reduce the manual effort for debugging and identifying the change that caused a test to fail.

5.3.5. Threats to Validity

In this section, the internal and external threats to the validity of this study are explained. Also, described are the precautions we took to limit the impact of the threat.

5.3.5.1. Internal Validity

We used PARTE tools for this experiment. A defect in those tools can affect the outcome of this experiment. We thoroughly examined the tools, and manually validated that the test repairs for simple programs were being performed correctly.

5.3.5.2. External Validity

We list three threats to external validity.

1. *Application Representativeness*: Applications used for our experiment do not represent all the available PHP web applications in terms of size, type, functionality, accessibility, user diversity etc. We minimized this threat by using real-world, open source, non-trivial, web-based PHP applications. We think this threat can be further mitigated when a larger variety including enterprise grade applications are evaluated using our approach.

2. *Test Representativeness*: We used unit tests in this experiment. In my experience at Microsoft, I have seen comparable quality of unit tests used to test commercial software. Though this improves the confidence in our approach, we think a variety of tests such as performance, stress and accessibility should be used to ensure high quality in an industrial software.

3. *Fault Representativeness*: We augmented real faults in the application with mutation faults. There is evidence that for the purpose of empirical studies, mutation faults can be representative of real faults [4, 69]. However, the distribution of the mutation faults may not be the same as real faults. Hence, we think further experimentation with real faults is needed to address this threat.

5.3.6. Discussions

Results of this experiment provided a strong evidence that the proposed approach can repair significant number of obsolete tests.

In Table 5.7, it can be observed that the highest and the lowest percentage of obsolete tests repaired was 100% and 25%, respectively. A fourth of the identified obsolete tests were repaired automatically. This means the manual effort that would have gone into debugging, locating and fixing the obsolete test is eliminated using our proposed approach.

The changes to the parameters in a method signature, is the only factor that appeared to have an effect on the percentage of obsolete tests repaired. We observed that when the reusable input value rate is higher, then the number of reusable tests is likely to be higher. This would result in a lower number of obsolete tests, which means there will be a fewer number of tests to repair. In *Mambo* version pair 4.5.5 and 4.5.6 input value reuse rate is 60% (refer to Table 5.2). The percentage of reusable tests relative to the selected tests was 68%, which is the highest obtained among all the applications and versions evaluated in the study. Percentage of obsolete tests repaired for this version pair was 78%, which is not the highest obtained but is higher than the average.

A change to a web application, either motivated by a customer request or to address a security concern, may require breaking the backward compatibility. A breaking change [98] can introduce a new argument in an existing function. An existing test that covers this function will fail since the test does not provide an input value for the new parameter. The scope of our test approach is limited to identifying the new argument. The repaired version of the test will include a comment that helps the developer to identify the change and determine an appropriate input value for the argument. In *Composer* version 1.4.0, a new argument '*ignoreFilters*' was introduced in the '*archive*' method. Our approach identifies the new parameter but the appropriate value i.e., '*$true*' or '*$false*' needs to be determined by the developer. Nevertheless, from my experience, having tools that can automatically detect such changes, can greatly improve productivity in an industrial setting. Additionally, automatically preparing a code submission so that the developers can review the submission and either approve or reject the changes to the test, will greatly minimize the manual effort and human errors.

6. DISCUSSION

The results from the three experiments clearly demonstrate that the proposed techniques can reduce the overall cost of regression testing of web applications. The results from the first experiment show that all version pairs of the applications have several input values that can be reused. As presented in Table 5.4, the highest number of reusable input values 19, 536, 708, 702 and 861, and the lowest number of reusable input values were 15, 57, 195, 219 and 280 respectively for *FAQForge*, *Mambo*, *Mantis*, *osCommerce* and *phpScheduleIt*. Compared to the control technique (baseline), which is the original PARTE approach, results from the current experiment demonstrates that every version pair has values that are reusable. The results from the second experiment show that utilizing existing test cases with input value reuse can produce further cost savings for regression testing. Considering the number of obsolete tests filtered from the selected tests, we believe that one can save a substantial amount of the human effort needed to manually inspect the tests by finding them automatically. Further, our results also showed that we can achieve cost savings by selecting a smaller set of test cases without sacrificing fault detection ability.

In the remainder of this section, we discuss the practical implications of these results and lessons learned from our experiments.

6.1. Cost Effectiveness of Input Value Reuse

Typically, the test paths require actual inputs to create executable test cases; thus, incorporating the input value reuse approach to PARTE that we introduced in our previous work can reduce the regression testing effort substantially. For example, the number of regression test paths for Mantis version 1.2.0 is 2,802. The number of input values needed for these paths is 20,425. For those test paths, our previous study [25] indicated that many constraints were not solvable by automatic resolvers. This means that many input values had to be resolved manually, which typically would require a tremendous effort. By automatically identifying the reusable input values for the modified versions, our approach can reduce a substantial manual effort. Note that we must manually resolve the input values that are not reusable from the previous versions or cannot be solved by automatic resolvers. This indicates that our approach can be beneficial for the application versions that contain a high number of reusable input values. Further, as constraint values remain applicable for multiple versions, our approach offers greater cost savings.

6.2. Relationship between Reuse Rates and Changes between Versions

The amount of changes between versions and the type of change can affect the input value reuse rate. For instance, *Mantis* version 1.2.0 underwent a major update that includes bug fixes and enhancements such as bug monitoring. There were many functional changes that affected the def-use of variables when compared to the earlier version 1.1.8. In the version pair, 1.1.7 and 1.1.8, there were many changes made to support localization features, and most of these modifications were made in library files or in resource strings needed for language translation. Typically, these types of changes do not affect the def-use of variables, and thus number of reusable input values was higher.

The relationship between the reuse rate and the code changes between the version pairs can be observed across all applications. For the version pair 1.3.0 and 1.3.1 in *FAQForge*, only one library file was modified. Among the 25 input values that are required to execute 5 regression paths, 19 values were reusable. A manual code inspection of *Mambo* version 4.6.1 showed that this version had several modifications such as the integration of a new editor, security fixes, and a couple of feature upgrades. Because of these changes, several input values could not be reused. In the two version pairs analyzed in *osCommerce*, 2.2MS2 had more functional changes than 2.2MS2-060817. Therefore, the number of reusable input values for 2.2MS2 was relatively lower. *phpScheduleIt* version 1.2.12 had a minor update that added a couple of localization features and bug fixes. Unlike this version, other two versions, 1.1.0 and 1.2.0, had major updates with feature upgrades and customizations, which produced a lower input reuse rate than version 1.2.12.

From these results, we learned that application versions with fewer changes or with changes introduced in small patches are more beneficial with respect to reducing test case generation costs than versions with many changes. Further, we believe that, irrespective of the number of changes in program source code, reusing constraints reduced the new test case generation effort by a substantial amount.

6.3. Relationship between Savings by Test Selection and Code Changes

The final number of tests selected in each version pair of an application depends on the coverage of existing unit tests, and the number and type of code changes between the versions. For instance, *Mantis* version 1.2.4 introduced major enhancements to a couple of features, and included fixes for various bugs reported in version 1.2.3. More than half of the existing tests covered the code paths that had undergone modifications and hence were selected as regression tests. However, given the type of code changes, the def-use of some variables between the versions were modified. Existing tests that provided input values to

such variables cannot be reused for regression testing, and hence these tests were marked as obsolete. Thus, only 8 of the existing tests remained in the final set of selected regression tests.

Similarly, between *Mambo* versions 4.6.3 and 4.6.4, there was much code churn due to refactoring and security bug fixes. Given the test coverage of existing tests in 4.6.3, the final number of tests selected was lower than for the other version pairs of this application. In version pairs investigated for *phpScheduleIt* and *log4php*, there were fewer changes, and most of those were non-functional. Thus, the final number of tests selected remained relatively lower.

From these results, we learned that irrespective of existing test coverage, and the number or type of changes between versions, regression test selection costs are further minimized when obsolete tests are automatically detected. Although our selection approach is effective in reducing costs, we also understand that there will be up-front costs of setting up the infrastructure to implement the test selection technique, and automating the process in an industrial setting. However, once these are automated, the savings from our approach will accrue over time.

To our knowledge, our studies are the first attempt to investigate the reusability of variable constraints and their input values, and to select tests such that obsolete tests are identified and automatically repaired. The proposed approaches produced promising results and the findings from the studies provide insight about how reusable constraint values can be utilized during the testing and regression testing process. Additionally, our studies showed how the test selection process can be improved by identifying obsolete tests using the constraint value reuse algorithm, and how the obsolete tests can be repaired automatically.

7. CONCLUSIONS AND FUTURE WORK

7.1. Merit and Impact

This research is expected to provide two benefits. It provides guidance on how to reuse existing resources in regression testing of a web application. It shows how to perform future research on improving a currently available regressing test technique by augmenting the technique with information about existing test artifacts. Researchers and software practitioners who follow our guidance can have a significant impact on reducing the overall cost of regression testing of their web applications.

First, this research offers guidance in exploring cost-effective techniques for regression testing of web applications. Our research demonstrates how existing resources i.e., tests and their input values can be reused. It shows how obsolete tests can be repaired and reused effectively in reducing the costs of regression testing of web applications.

Second, this research demonstrates how an existing cost-effective technique can be further improved by augmenting it with information from other test techniques. We used the PARTE approach of minimizing regression test paths as a foundation to build additional components such as *Reusable Tests* and *Test Repair*. We demonstrated how these components can be used on real-world applications to reduce the costs of regression testing by reusing existing resources. Our approach shows a path for researchers and software practitioners to further explore cost-effective techniques for regression testing of web applications.

7.2. Future Directions

In this research, we presented an approach to reducing the effort required to conduct regression testing of web applications, which consists of three techniques: reusable constraint value identification, test selection and test repair. We performed three controlled experiments to evaluate the proposed approach. Each experiment used non-trivial, open-source PHP web applications with multiple versions. Our results indicate that the approach can save substantial effort when testing new versions of an application. The results found that a large portion of constraint values can be reused, a small number of existing tests can be selected to test new versions, and some of the obsolete tests among the selected tests can be repaired automatically. Thus, we can significantly reduce the effort required to test new versions of an application by using these three techniques. Further, the constraint values for variables may be reusable across several versions if the definition and use relationships of the variables hold across the versions, and the maintenance overhead for

test cases can be reduced by selecting a subset of test cases across several versions. This means that we can expect greater savings as the applications evolve over time.

The results of our studies suggest several avenues for future work. First, we evaluated our approach using several widely used open-source web applications with multiple versions. While the experiment results are promising, future studies should investigate whether our approach can be applied to an industrial setting where there are many other constraints such as a larger number of tests per version, diverse testing practices used by organizations and the development/testing environment. Further, the experiment results we obtained in this study do not sufficiently capture the potential benefits and factors related to the long-term utilization of our techniques. We believe that our approach would provide greater benefits if we were to apply it over a long period of time. It would be interesting to examine whether certain variables are more sustainable over several version changes and to investigate plausible reasons (e.g., certain usage patterns associated with the variables) including measuring the human effort required for manually resolving input constraint values.

For commercial web applications, fixing critical security bugs in a short span of time is crucial because delaying patch releases can lead to a significant loss for companies. Thus, in addition to selecting a smaller set of test cases with reusable values, identifying more important test cases (e.g., test cases covering risky components or high-modification areas) early could shorten the regression testing time. Thus, we plan to investigate hybrid techniques that combine test selection with test case prioritization to explore such potentials.

REFERENCES

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[2] Nadia Alshahwan and Mark Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 45–55, Minneapolis, MN, USA, 2012. ACM.

[3] Jeffrey R. Anderson. *Understanding Contextual Factors in Regression Testing Techniques*. Phd thesis, North Dakota State University, Fargo, ND, USA, 2016.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, St. Louis, MO, USA, 2005. ACM.

[5] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, TAIC-PART '06, pages 137–146, 2006.

[6] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 292–301. IEEE, 1993.

[7] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 265–274. ACM, 2010.

[8] Microsoft Dynamics AX. `https://dynamics.microsoft.com/en-us/ax-overview/`. Accessed October 2018.

[9] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 433–444, Gothenburg, Sweden, 2018. ACM.

[10] R. Boggs, J. Bozman, and R. Perry. Reducing downtime and business loss: Addressing business risk with effective technology. Technical report, International Data Corporation (IDC), 2009. Accessed October 2018.

[11] Haipeng Cai, Raul Santelices, and Douglas Thain. Diapro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *ACM Trans. Softw. Eng. Methodol.*, 25(2):18:1–18:50, April 2016.

[12] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, Honolulu, USA, 2011. ACM.

[13] Yanping Chen, Robert L. Probert, and Hasan Ural. Model-based regression test suite generation using dependence analysis. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, A-MOST '07, pages 54–62. ACM, 2007.

[14] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 24–29. ACM, 2011.

[15] Composer web page. `https://github.com/composer/composer/`. Accessed October 2018.

[16] Composer issue 3338. `https://github.com/composer/composer/issues/3338#issuecomment-212841454/`. Accessed October 2018.

[17] Composer pull request 5174. `https://github.com/composer/composer/pull/5174/`. Accessed October 2018.

[18] Composer pull request 5224. `https://github.com/composer/composer/pull/5224/`. Accessed October 2018.

[19] Composer commit dac51c7f4b1d58d2781fa29e3becce7bb3cb70dc. `https://github.com/composer/composer/commit/dac51c7f4b1d58d2781fa29e3becce7bb3cb70dc/`. Accessed October 2018.

[20] Composer commit ff46816e798fa16691f6ef2fb23f7f7b700c7670. https://github.com/composer/composer/commit/ff46816e798fa16691f6ef2fb23f7f7b700c7670/. Accessed October 2018.

[21] Johannes Dahse and Thorsten Holz. Experience report: An empirical study of php security mechanism usage. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 60–70. ACM, 2015.

[22] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: A tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 11–15, Beijing, China, 2012. ACM.

[23] Brett Daniel, Danny Dig, Tihomir Gvero, Vilas Jagannath, Johnston Jiaa, Damion Mitchell, Jurand Nogiec, Shin Hwei Tan, and Darko Marinov. Reassert: A tool for repairing broken unit tests. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1010–1012, Waikiki, Honolulu, HI, USA, 2011. ACM.

[24] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 78–87. ACM, 2005.

[25] Hyunsook Do and Md. Hossain. An efficient regression testing approach for php web applications: a controlled experiment. *Journal of Software Testing, Verification and Reliability*, 24:367–385, 2014.

[26] Ravi Eda, Hyunsook Do, and Md. Hossain. An efficient regression testing approach for php web applications using reusable constraints, test selection and test repair. *Information and Software Technology*, 2018. In review.

[27] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338. IEEE, 2001.

[28] Emelie Engström, Mats Skoglund, and Per Runeson. Empirical evaluations of regression test selection techniques: A systematic review. In *Proceedings of the Second ACM-IEEE International*

Symposium on Empirical Software Engineering and Measurement, ESEM '08, pages 22–31. ACM, 2008.

[29] L. Eshkevari, F. Dos Santos, J. R. Cordy, and G. Antoniol. Are php applications ready for hack? In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 63–72, Montreal, Canada, 2015. IEEE.

[30] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954. ACM, 2009.

[31] Xiaocheng Ge, Richard F. Paige, Fiona A.C. Polack, Howard Chivers, and Phillip J. Brooke. Agile development of secure web applications. In *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06, pages 305–312, Palo Alto, California, USA, 2006. ACM.

[32] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 430–440. IEEE, 2012.

[33] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418. IEEE, 2009.

[34] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 285–296. ACM, 2009.

[35] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 751–762. ACM, 2016.

[36] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. Is this a bug or an obsolete test? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 602–628, Montpellier, France, 2013. Springer-Verlag.

[37] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. *SIGSOFT Softw. Eng. Notes*, 18(3):160–170, July 1993.

[38] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Florence, Italy, 2015. IEEE.

[39] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 377–386. ACM, 2010.

[40] Md. Hossain, Hyunsook Do, and Ravi Eda. Regression testing for web applications using reusable constraint values. In *Proceedings of the Fourth International Workshop on Regression Testing (Regression 2014)*, pages 312–321, Cleveland, USA, 2014.

[41] Humbug: Mutation testing for php. `https://github.com/humbug/humbug/`. Accessed October 2018.

[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, Hong Kong, 2014. ACM.

[43] Shaikh Jeeshan Kabeer, Maleknaz Nayebi, Guenther Ruhe, Chris Carlson, and Francis Chew. Predicting the vector impact of change: An industrial case study at brightsquid. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, pages 131–140. IEEE, 2017.

[44] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv.*, 50(2):29:1–29:32, May 2017.

[45] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, February 2013.

[46] Mijung Kim, Jake Cobb, Mary Jean Harrold, Tahsin Kurc, Alessandro Orso, Joel Saltz, Andrew Post, Kunal Malhotra, and Shamkant B. Navathe. Efficient regression testing of ontology-driven systems. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 320–330. ACM, 2012.

[47] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[48] Jung-Hyun Kwon, In-Young Ko, and Gregg Rothermel. Prioritizing browser environments for web application test execution. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 468–479, Gothenburg, Sweden, 2018. ACM.

[49] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 821–830. ACM, 2017.

[50] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 583–594, Seattle, WA, USA, 2016. ACM.

[51] H. K. N. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69. IEEE, Oct 1989.

[52] Yuan-Fang Li, Paramjit K. Das, and David L. Dowe. Two decades of web application testing-a survey of recent advances. *Inf. Syst.*, 43(C):20–54, July 2014.

[53] log4php web page. `http://logging.apache.org/log4php/`. Accessed October 2018.

[54] log4php issue 133. `https://issues.apache.org/jira/browse/LOG4PHP-133/`. Accessed October 2018.

[55] Mambo web page. `http://mambo-foundation.org/`. Accessed October 2018.

[56] Faqforge web page. https://sourceforge.net/projects/faqforge/. Accessed October 2018.

[57] Mantis web page. http://www.mantisbt.org/. Accessed October 2018.

[58] oscommerce web page. https://www.oscommerce.com/. Accessed October 2018.

[59] Aaron Marback, Hyunsook Do, and Nathan Ehresmann. An effective regression testing approach for php web applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 221–230, Montreal, Canada, 2012.

[60] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458. IEEE, 2015.

[61] Atif M. Memon and Mary Lou Soffa. Regression testing of guis. *SIGSOFT Softw. Eng. Notes*, 28(5):118–127, September 2003.

[62] S. Mirarab, S. Akhlaghi, and L. Tahvildari. Size-constrained regression test case selection using multicriteria optimization. *IEEE Transactions on Software Engineering*, 38(4):936–956, 2012.

[63] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezze. Supporting test suite evolution through test case adaptation. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 231–240, Montreal, Canada, 2012. IEEE.

[64] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018.

[65] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing, 3rd Edition*. John Wiley & Sons, Nov 2011.

[66] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, San Francisco, USA, 2013. IEEE.

[67] Jesper Öqvist, Görel Hedin, and Boris Magnusson. Extraction-based regression test selection. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on*

the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '16, pages 5:1–5:10, Lugano, Switzerland, 2016. ACM.

[68] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132. ACM, 2014.

[69] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 537–548, Gothenburg, Sweden, 2018. ACM.

[70] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, May 2014.

[71] phpscheduleit web page. `http://phpscheduleit.org/`. Accessed October 2018.

[72] phpscheduleit issue 18. `https://sourceforge.net/p/phpscheduleit/bugs/18/`. Accessed October 2018.

[73] Phpunit - official site. `http://phpunit.de/`. Accessed October 2018.

[74] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 33:1–33:11. ACM, 2012.

[75] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165. ACM, 2014.

[76] The redmonk programming language rankings: June 2018. `https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/`. Accessed October 2018.

[77] Microsoft Visual Studio release rhythm. `https://docs.microsoft.com/en-us/visualstudio/productinfo/vs2017-release-rhythm/`. Accessed October 2018.

[78] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, October 2004.

[79] 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, 2001.

[80] Filippo. Ricca and Paolo. Tonella. Testing processes of web applications. *Ann. Softw. Eng.*, 14(1-4):93–114, December 2002.

[81] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[82] Gregg Rothermel and Mary Jean Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 201–210. IEEE, 1994.

[83] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[84] Konstantin Rubinov and Jochen Wuttke. Augmenting test suites automatically. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1433–1434. IEEE, 2012.

[85] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 277–287. IEEE, 2012.

[86] Raul Santelices and Mary Jean Harrold. Applying aggressive propagation-based strategies for testing changes. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 11–20. IEEE, 2011.

[87] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 237–247. ACM, 2015.

[88] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, ISSRE '12, pages 311–320. IEEE, 2012.

[89] Xiaobing Sun, Bixin Li, Hareton Leung, Bin Li, and Junwu Zhu. Static change impact analysis techniques. *J. Syst. Softw.*, 109(C):137–149, November 2015.

[90] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: Guided path exploration for efficient regression test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 1–11. ACM, 2011.

[91] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Prato, Italy, 2008. Springer-Verlag.

[92] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, June 2003.

[93] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11. ACM, 2012.

[94] Microsoft Visual Studio. `https://visualstudio.microsoft.com/vs/`. Accessed October 2018.

[95] Usage of server-side programming languages for websites. `https://w3techs.com/technologies/overview/programming_language/all/`. Accessed October 2018.

[96] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 249–260. ACM, 2008.

[97] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE, 1981.

[98] Laerte Xavier, Aline Brito, Andre Hora, and Marco. T. Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, SANER '17, pages 138–147, Klagenfurt, Austria, Feb 2017. IEEE.

[99] Lei Xu, Baowen Xu, and Jixiang Jiang. Testing web applications focusing on their specialties. *SIGSOFT Softw. Eng. Notes*, 30(1):10–, January 2005.

[100] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. A hybrid directed test suite augmentation technique. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 150–159. IEEE, 2011.

[101] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 257–266. ACM, 2010.

[102] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[103] Celal Ziftci and Jim Reardon. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at google scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 113–122, Buenos Aires, Argentina, 2017. IEEE.