EFFICIENT REGRESSION TESTING FOR WEB APPLICATIONS USING REUSABLE CONSTRAINT VALUES

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

By

Md Imamul Hossain

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

May 2013

Fargo, North Dakota

North Dakota State University

Graduate School

Title

Efficient Regression Testing for Web Applications
Using Reusable Constraint Values

By

Md Imamul Hossain

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Hyunsook Do
Chair

Dr. Saeed Salem

Dr. Mukhlesur Rahman

Approved:

05/07/2013	Dr. Brian M. Slator
Date	Department Chair

ABSTRACT

Current web applications offer people easy ways to deploy web sites, and the use of web applications has grown rapidly over the past decades. Companies that provide web applications need frequent regression testing because of various security attacks and frequent feature update demands from users. Typically, such applications require regression testing with a short turn-around time because they have already been deployed and used in the field. Recent research has presented an efficient regression testing approach that allows us to focus on the areas of code that have been changed. But, this approach requires a time consuming process of constraints resolution. Here, a technique has been presented that identifies reusable constraint values from the previous version to execute regression test paths for the new version. Also, the empirical study shows that a significant reuse can be achieved by this technique which reduces the overall time of regression testing.

ACKNOWLEDGMENTS

I would like to thank the advisory committee members and professors at North Dakota State University (NDSU), Dr. Hyunsook Do, Dr. Saeed Salem and Dr. Mukhlesur Rahman for their valuable support and guidance. As my advisory committee chair, Dr. Hyunsook Do always made time in providing technical expertise and guidance throughout the process. This would not have been possible without her constant support.

I am also thankful to the National Science Foundation for funding this research.

TABLE OF CONTENTS

ABSTRACT……………………………………………………………………………………...iii

ACKNOWLEDGMENTS…………………………………………………………………......iv

LIST OF TABLES…………………………………………………………………………vii

LIST OF FIGURES……………………………………………………………………….viii

CHAPTER 1. INTRODUCTION ………………………………………………….............1

CHAPTER 2. BACKGROUND AND RELATED WORK …………………………...............3

CHAPTER 3. METHODOLOGY …………………………………………….....................5

3.1. Collecting Reusable Constraint Values ………………………………………..7

3.2. An Example of Reusable Input Variable Identification …………………………11

CHAPTER 4. EMPIRICAL STUDY ………………………………….........................15

4.1. Objects of Analysis ……………………………………………………………...15

4.2. Variables and Measures …………………………………………………………18

4.3. Experiment Setup ………………………………………………………………..18

4.4. Threats to Validity ………………………………………………………………19

CHAPTER 5. DATA AND ANALYSIS ………………………………...........................20

5.1. Results for FAQForge …………………………………………………………...22

5.2. Results for osCommerce ………………………………………………………...26

5.3. Results for Mambo ………………………………………………………………28

5.4. Results for Mantis ………………………………………………………………29

5.5. Results for phpScheduleIt ………………………………………………………31

CHAPTER 6. DISCUSSION ………………………………..34

CHAPTER 7. CONCLUSION ………………………………..37

REFERENCES ……………………………..…………………………………..……39

LIST OF TABLES

Table Page

1. Three Versions of PHP Sample Program ……….……………………………………12
2. Test Path ……………………………………………………………………………13
3. Constraints …………………………………………………………………………13
4. Objects of Analysis ………………………………………………………………..17
5. Total Number of Paths and Regression Paths ………………………………………..21
6. The Number of Input Values (Total vs.Reusable)……………………………….....24
7. Input Values Reuse Rates ……………………….…………………………........25

LIST OF FIGURES

Figure | Page

1. Overview of PARTE [1] and Constraint Reuse ……….………………………………….…...6

2. Algorithm FindReusableInputValues ……………………………………………….......10

3. Algorithm CheckStatementSimilarity ……………………………………………….......11

4. FAQForge Input Values Comparison ………………………………………………...22

5. FAQForge Reuse Rates ………………………………………………………………….23

6. osCommerce Input Values Comparison ……………………………………………...26

7. osCommerce Reuse Rates ………………………….…………………………………27

8. Mambo Input Values Comparison ……………………………………………………28

9. Mambo Reuse Rates ……..………………………………………………………..........29

10. Mantis Input Values Comparison ……………………………………………………30

11. Mantis Reuse Rates …………………………………………………………………...31

12. phpScheduleIt Input Values Comparison ……………………………………………32

13. phpScheduleIt Reuse Rates …………………………………………………………...32

CHAPTER 1. INTRODUCTION

The use of web applications has grown rapidly over the past decade, and a large number of companies have relied heavily on web applications for their businesses. Companies that provide web applications often encounter various security attacks and frequent feature update demands from users, and when they do, companies need to fix security problems or upgrade the application with new features. Thus, such applications undergo frequent patch releases which require frequent regression testing processes that can support a short turn-around time in releasing patches instead of applying regression testing to the entire product.

Recently, a new regression testing approach PARTE (PHP Analysis and Regression Testing Engine) [1] was introduced which allows us to focus on the areas of code that have been changed and to regression test them. In particular, a technique was developed that generates test cases using program slices and their inputs that require constraint resolution. The experiment results with five open web applications indicated that the approach is efficient in reducing the cost of regression testing by reducing the number of test cases to exercise. It was also learned that resolving input constraints was a time consuming process because many input constraints required manual resolution even after applying the external constraint resolution tools. Further, automatic constraint solvers can take a long time to resolve constraints for some inputs (e.g., long strings) [2], [3]. If input constraints and their resolved values from the previous version's test cases can be identified which are applicable to the current version, greater savings can be expected.

Here, a technique is proposed that identifies reusable constraint values for regression test cases (both new test cases and selected test cases from the previous version) to accommodate further savings with regression testing for web applications that require frequent patches and

short regression testing cycles. The technique finds variables where the input value from the previous version can be reused to execute the regression test path for the new version. By comparing definitions and uses of a particular variable between the old and new versions of the object program, it is determined whether the same constraints for the variable can be used. By doing this, unnecessary effort during regression testing can be avoided: this approach helps us avoid collecting constraints for the reusable variables as well as reduces the number of numeric and string inputs that need to be resolved.

To assess the approach, an empirical study has been designed and performed using five open source web applications. The result shows that a large number of variable constraints can be reused from the previous version's test cases, thus it can reduce a significant amount of effort for resolving constraint values for those variables when a new version of the application is tested. Further, because the constraints and actual values for variables can be reused across several versions as long as the defined conditions are satisfied, greater savings over time can be expected.

The thesis is organized as follows. In Chapter 2, some background as well as related work relevant to web applications and regression testing have been provided. In Chapter 3, the overall methodology is described, including a brief description of PARTE and the technical details about the proposed approach. Chapter 4 presents objects of analysis, variables and measures, experimental setup and threats to validity. Chapter 5 provides the data and analysis for the object programs. Chapter 6 discusses the rationale of the results for different version of object programs, and Chapter 7 presents conclusions and future work.

CHAPTER 2. BACKGROUND AND RELATED WORK

To date, researchers have studied various methods for improving the cost-effectiveness of regression testing, and most of them have focused on reusing the existing test cases (e.g., [9], [10], [11], [12]). However they are often insufficient to retest code or system behaviors that are affected by code changes. To address this problem, recently, researchers started working on test suite augmentation techniques which create new test cases for areas that have been affected by changes [13], [14], [15], [16], [17], [18] and [33]. Apiwattanapong et al. [13] and Santelices et al. [14] presented a propagation-based approach which uses program dependence analysis and symbolic execution to identify areas affected by code changes, and then provides test requirements for changed software. Xu et al. [15], [16] presented an approach to generate test cases by utilizing the existing test cases and adapting concolic and genetic test case generation techniques. Taneja et al. [17] proposed an efficient test generation technique that uses dynamic symbolic execution, eXpress, by pruning irrelevant paths. Chen, Probert and Ural et al. [18] proposed a model-based regression test suite generation using dependence analysis. Rubinov and Wuttke et al. [33] presented a framework for augmenting test suites automatically. Alshahwan and Harman et al. [32] proposed output diversity increase to make test suite augmentation more effective. These approaches focused on desktop applications such as C or Java, but the approach here applies regression testing to web applications, creating different challenges. Further, the major focus in this thesis is to identify reusable constraint input values by analyzing a variable's definition-use information.

In the area of web applications, several researchers proposed various test case generation approaches. Wassermann et al. [19] and Artzi et al. [20] utilized a concolic approach to generate test cases for PHP web applications. Other test case generation techniques for web applications

used crawlers and spiders to identify and test web application interfaces. For instance, Ricca and Tonella [21] used a crawler to discover the link structure of web applications and to produce test case specifications from this structure. Deng et al. [22] used static analysis to extract information related to URLs and their parameters for web applications, and to generate executable test cases using the collected information. Halfond et al. [23] presented an approach that uses symbolic execution to identify precise interfaces for web applications. While this focuses on generating test cases for the areas affected by code changes and collecting reusable constraint input values, the aforementioned approaches for web applications did not consider regression testing aspects. Some work on regression testing for web applications [24], [25] has been done before, but their focus was different than this work. Dobolyi and Weimer [24] presented an approach that automatically compares the outputs from similar web applications to reduce the regression testing effort. Elbaum et al. [25] presented a web application testing technique that utilizes users' session data considering regression testing contexts.

Another research area that is slightly relevant to our work is constraint resolution. Many researchers have worked on this area [2], [26], [27], [28], and now, several constraint resolution tools are available, including Hampi [5] and Choco [6], which have been used in recent work [1]. Although these automatic resolution tools helped resolving many input values (in particular, numeric values), manual inspection and resolution are required for many of them due to the complexity of inputs and the time required to resolve them, which motivated this research (collecting reusable constraint input values). By combining these two approaches (automatic resolution using constraint solvers and the approach proposed in this thesis), great effort savings can be expected during regression testing.

CHAPTER 3. METHODOLOGY

As presented in the paper [1], PARTE is implemented to generate new regression test cases by analyzing the impacted areas by changes. To facilitate the approach proposed in this thesis, PARTE is extended as shown in Figure 1. The dark gray area is the new addition to PARTE. To provide an overview of the approach, PARTE's main components are summarized and then the components related to the constraint resolution reuse are explained.

In Figure 1, the boxes depict the main activities; the ovals depict the inputs and outputs of the processes; and the double ovals represent the final outcomes of the approach. PARTE has three main components: (1) Preprocessing - Before the impact analysis is performed on PHP web applications, a preprocessing step is required to handle dynamic aspects of PHP web applications and to preserve variable names across versions as PHP compiler, PHC [4] which is used here does not handle these (see [1] for details). (2) Impact analysis - Based on preprocessed files, PARTE generates program dependence graphs (PDGs) for two consecutive versions of the PHP web application, and generates program slices using code change information. (3) Test case generation - PARTE generates new test cases for the impacted areas of code by using program slices and considering both string and numeric input values. To resolve input constraint values, two constraint solvers, Hampi [5] and Choco [6] are used. Note that they are included in the test case generation box, but they are not part of PARTE. (They are external constraint solvers.)

To apply this approach, existing test paths and executable test cases that have been used for testing the previous version are used. The database for version v0 at the bottom right corner of Figure 1 contains these two sets of information. To collect reusable constraints and input values from the previous version, the following steps are required: (1) First, the test paths for the new version are generated. To do so, two consecutive versions of PHP files are analyzed to

identify program slices by identifying code changes, and then, test paths that are required for the new version are generated. (2) Two sets of compatible test paths (the previous and current versions) are compared to collect the same variables that are used in both versions. Then, the constraints for those variables and the corresponding input values that can be reused for the new paths are identified by analyzing variable definitions and uses.

These two steps are described in detail in the following subchapters, including an algorithm and an example that show how the approach works.

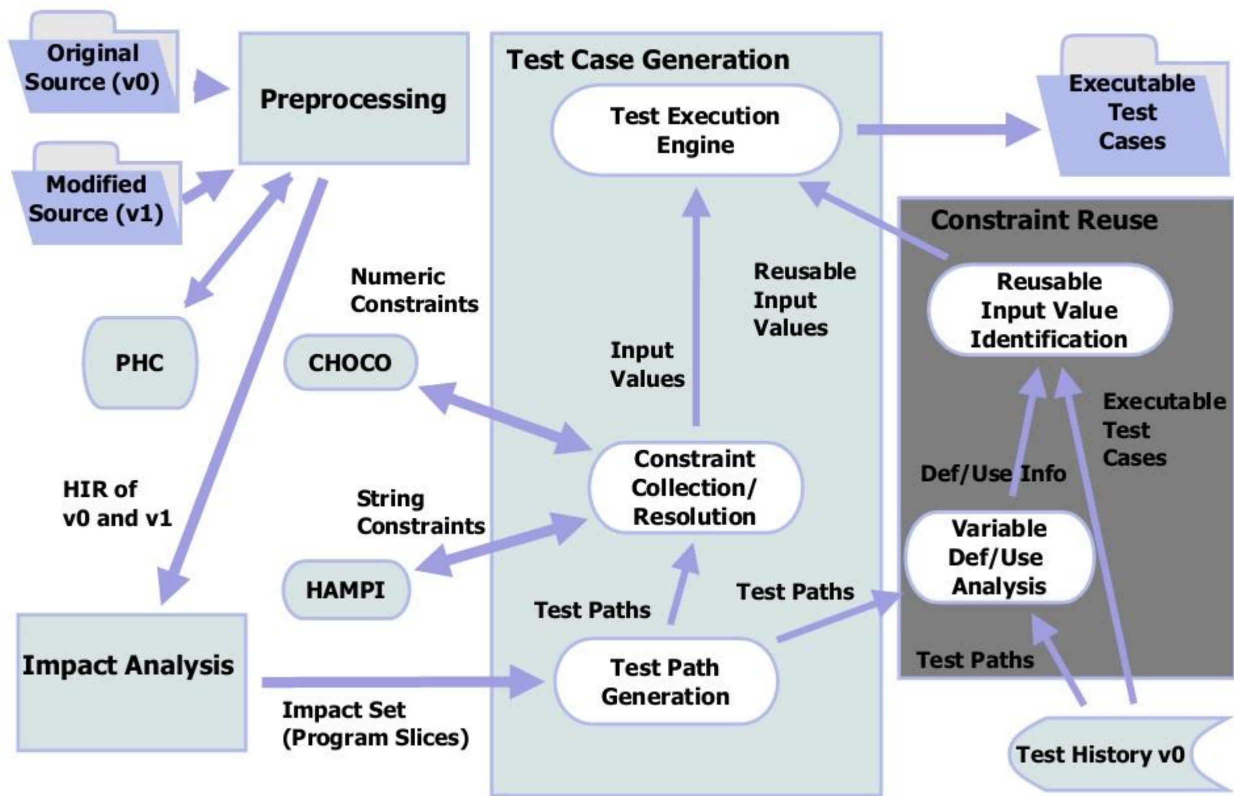

Figure 1. Overview of PARTE [1] and Constraint Reuse

3.1. Collecting Reusable Constraint Values

Without considering the use of reusable constraint input values, a typical way to generate executable test cases using PARTE, as follows. Once all necessary preprocessing activities are done as explained earlier, the path generator creates test paths for the new version using program slices obtained by analyzing two consecutive versions of the PHP web application. To execute the test paths, actual input values are needed to assign for the input variables. To do so, the constraint collector gathers the constraints for these input variables, and then, the constraint resolution tool generates the input values that satisfy the constraints. (Input values are needed to resolve manually if the tool cannot resolve them.) However, as it is briefly mentioned in Chapter 1, resolving input constraints takes a lot of time and effort, depending on the number of changes and the complexity of input constraints. For web applications with small patches, the number of changes in the new version is often very small, but sometimes, even with small changes, the impacted areas by changes could be large, thus the number of inputs could be large. To address this issue, a technique was implemented that collects reusable input constraint values from the executable test cases of the previous version.

As shown in Figure 1, the proposed technique consists of two processes (right, dark gray box): variable def/use analyzer and reusable input value identifier. The variable def/use analyzer reads the test paths for the new version generated by the path generator and the test paths from test history database for the previous version. Although the figure does not show it, the def/use analyzer also needs to read the PDG files because the test paths generated by the path generator do not provide variable information, but PDGs contain definition and use information for variables. The analyzer finds the information and builds a variable mapping table that contains a list of block numbers, and definitions and uses of the variables appeared on the test path. The

PDGs are constructed based on the basic blocks. The reusable input value identifier reads the mapping table and executable test cases of the previous version, assigning values to the reusable variables by extracting input values from the previous version's executable test cases.

The process explained here is formally shown in the *FindReusableInputValues* algorithm (Figure 2). The algorithm takes four inputs: old test path, new test path, old PDG, and new PDG. Here, old indicates the previous version, and new indicates the current version under test. The algorithm produces one output: reusable variables with actual values. In line 1, a new map is declared; the map maps variable names to the list of PDG nodes that contains definitions and uses of that particular variable for the new path. For each node on the new path, the corresponding PDG node is loaded from the new PDG, and the block id is found from the path node (line 3). If the current node has variable with definition and/or uses, the map is updated by calling *UpdateDefUseMap* which changes or inserts the variable information (line 4). A map is also defined for the old path following the same procedure (lines 9 to 14). At this point, all def/use maps are created for both the old and new versions.

Next, the algorithm finds reusable input values for the variables it found in the previous step. For each candidate variable in the new def/use map, the corresponding PDG nodes are extracted. Each PDG node contains actual source code statements, so the algorithm extracts those statements from the node. If the variable is found in the old def/use map, then the source code statements are extracted in the same way.

The source code statements between two versions (old and new) are compared using a *CheckStatementSimilarity* function (line 19). If the statements match, then the variable is identified as reusable and added to the reusable variables list (line 21).

CheckStatementSimilarity function takes current variable and def/use map of both old path and new path. The function returns true or false depending on the matching logic. In line 1, it collects only those statements which contain definition and use of the input variable on the old path. In line 2, the same information is collected for the new path. After that, all of the statements in the new path are iterated one by one in line 3. A complete match is calculated for each statement, by comparing corresponding statement in the old path. This is a character by character matching and is performed in line 4. If there is a mismatch then the algorithm proceeds to find a partial match in line 5.

A partial match would occur if the entire statement does not match, but the definition or use portion of a variable in the statement matches exactly with the old version. For instance, one statement in the old version is written as “if (a > 3),” and the statement is changed to “if (a > 3 || b == 1)” in the new version. Because the new statement contains the use of variable “a” with the same condition i.e. (a > 3), it is considered to be a partial match and the variable “a” is added to the reusable variable list.

If both complete and partial match fail, then false is returned in line 6, which means the variable is not eligible to reuse. Otherwise, the algorithm proceeds with variable dependency check in line 9. This checks whether the input variable has dependency on any other variable in the statement. An input variable is detected to have dependency if any definition or use of the input variable contains other variables rather than just string literals or constants. If a dependency is detected then the algorithm tries to resolve the dependency in line 10.

A dependency is resolved if all of the variables upon which the input variable is dependent returns true on *CheckStatementSimilarity*. During this secondary checking, cyclic dependency is avoided by excluding the path statement where the dependency was detected.

```
---------------------------------
Algorithm FindReusableInputValues
---------------------------------

Inputs: oldPath, newPath, oldPdg, newPdg
Outputs: reusable variables

1. newPathDefUseMap -> mapping variable to Def/Use statement
2. for n <- 0, n < newPath.size(), n++ do
3.     curBlock <- newPdg.getBlock(newPath.getBlockID(n))
4.     if (curBlock.HasDefOrUse())
5.         UpdateDefUseMap(curBlock, newPathDefUseMap)
6.     end if
7. end for

8. oldPathDefUseMap -> mapping variable to Def/Use statement
9. for n <- 0, n < oldPath.size(), n++ do
10.    curBlock <- oldPdg.getBlock(oldPath.getBlockID(n))
11.    if (curBlock.HasDefOrUse())
12.        UpdateDefUseMap(curBlock, oldPathDefUseMap)
13.    end if
14. end for

15. reUsableVars -> list of reusable variables
16. for n <- 0, n < newPathDefUseMap.getVars.size(), n++ do
17.     curVar <- newPathDefUseMap.getVars()[n]
18.     if (oldPathDefUseMap.getVars().contains(curVar))
19.         isSimilar <- CheckStatementSimilarity(curVar, oldPathDefUseMap,
                                              newPathDefUseMap)
20.         if (isSimilar)
21.             reUsableVars.Add(curVar)
22.         end if
23.     end if
24. end for
25. return reUsableVars
```

Figure 2. Algorithm FindReusableInputValues

```
---------------------------------
Algorithm CheckStatementSimilarity
---------------------------------

Inputs: curVar, oldPathDefUseMap, newPathDefUseMap
Outputs: true/false

1. oldPathStmnts <- oldPathDefUseMap.getStatements(curVar);
2. newPathStmnts <- newPathDefUseMap.getStatements(curVar);
3. for n <- 0, n < newPathStmnts.size(), n++ do
4.     if (!CompleteMatch(newPathStmnts[n],oldPathStmnts[n]))
5.         if (!PartialMatch(curVar,newPathStmnts[n],oldPathStmnts[n]))
6.             return false;
7.         end if
8.     end if
9.     if (HasVarDependency(curVar,newPathStmnts[n]))
10.         if (!ResolveVarDependency(curVar,newPathStmnts[n])))
11.             return false;
12.         end if
13.     end if
14. end for
15. return true;
```

Figure 3. Algorithm CheckStatementSimilarity

3.2. An Example of Reusable Input Variable Identification

In this chapter, it was illustrated how reusable input variables can be identified using an example. Suppose there are three consecutive versions of a simple PHP program (v0.php, v1.php, and v2.php) as shown in Table 1.

As the example shows, from version v0 to version v1, statement 5 has been changed (*$a = $a – 1* to *$a = $a – 3*). For this case, the path that needs to be regression tested is shown in the third row of Table 2. Executing this path requires constraints for the variables to be gathered and the actual input values to be resolved. The second column of Table 3 shows the variable constraints for version v1.

In the tables, information for version v0 was added as a reference. For version v0, there are more test paths than the one path that appeared in Table 2, but to simplify the explanation

with the example, only the path that is relevant to the regression test path for version v1 is shown. Also, the first column of Table 3 shows the variable constraints for version v0.

In Table 1, from version v1 to version v2, it can be seen that statement 4 has been changed *($b = 6* to *$b = $b −1*). This case also requires one test path to be executed as shown in the fourth row of Table 2. Again, the constraints for the variables on that path are shown in the third column of Table 3. From the second column of Table 3, it can be seen that the constraints for variable "a" are unchanged.

Table 1. Three Versions of PHP Sample Program

<table>
<tr><th>v0.php (original version)</th><th>v1.php (modified version)</th><th>v2.php (modified version)</th></tr>
<tr><td><pre>1. $a = $_POST['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4. $b = 6;
5. $a = $a-1;
   }
   else
6. $b = $b+3;
7. if ($a > 7)
   {
8.   if ($b == 5)
9.   echo "a\n";
       else
10. $b= 7;
   }
11. echo "b\n";
12. echo "done

processing\n";</pre></td><td><pre>1. $a = $_POST['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4. $b = 6;
5. $a = $a-3; //changed
   }
    else
6. $b = $b+3;
7. if ($a > 7)
   {
8.   if ($b == 5)
9.   echo "a\n";
       else
10. $b= 7;
   }
11. echo "b\n";
12. echo "done

processing\n";</pre></td><td><pre>1. $a = $_POST['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4. $b = $b-1; //changed
5. $a = $a-3;
   }
    else
6. $b = $b+3;
7. if ($a > 7)
   {
8.   if ($b == 5)
9.   echo "a\n";
       else
10. $b= 7;
   }
11. echo "b\n";
12. echo "done

processing\n";</pre></td></tr>
</table>

To find the reusable constraints for the variables and their values, it was needed to first identify variables used in both the old and new test paths.

In this example, the paths for all three versions, v0, v1, and v2, have two variables such as variable "a" and variable "b." All definitions of variable "a" are identified and stored in a map by analyzing the corresponding PDG as explained in the algorithm description. The definitions

of variable “a” on the paths for all three versions are statements 1 and 5. Once the definitions of variable “a” are collected, all uses of variable “a” are identified and stored in the map. In this case, the uses for variable “a” are statements 3, 5, and 7 for all the three versions.

Next, for variable “a”, all definition-use statements are gathered from the new version’s PDG and then compared with all the definition-use statements gathered from the old version (v0 with v1 and v1 with v2). In this example, the definition-use statements for variable “a” on the new regression path for both version pairs are 1, 3, 5, and 7. Now, it can be seen that statement 5 in the old version (v0.php) is different from the new version (v1.php). The constraints for variable “a” cannot be reused for generating input values for version v1. However, for the next version pair, v1-v2, the statements that define and use variable “a” are identical, so in this case, the constraints for variable “a,” including the input value for variable “a” can be reused.

Table 2. Test Path

Version	**Test Path**
v0	{1, 2, 3, 4, 5, 7, 8, 9, 11, 12}
v1	{1, 2, 3, 4, 5, 7, 8, 9, 11, 12}
v2	{1, 2, 3, 4, 5, 7, 8, 10, 11, 12}

Table 3. Constraints

v0	**v1**	**v2**
$a < 12 $a – 1 > 7 $b == 5	$a < 12 $a – 3 > 7 $b == 5	$a < 12 $a – 3 > 7 $b – 1 == 5

For variable “b,” the definition-use statements on the new regression path are 2, 4, and 8 for the version pair v0-v1. For this version pair, the statements that define and use variable “b” are identical, thus constraints for variable “b,” including input values for variable “b,” can be reused for regression test path execution of v1. For the next version pair, v1-v2, the definition-use statements on the new regression path are 2, 4, 8 and 10. The statements that define and use variable “b” are not identical for this version pair, constraints for variable “b,” including input values for variable “b,” cannot be reused for regression test path execution of v2.

CHAPTER 4. EMPIRICAL STUDY

The goal of the proposed approach is to reduce the overall effort for generating test cases by reusing input values. To assess the approach, the following research question was considered.

RQ: Can the approach be efficient in reducing efforts to generate new test cases during regression testing?

To address the research question, an empirical study is designed and performed. The following subchapters describe the objects of analysis, independent variables, dependent variables and measures, study setup and design, and threats to validity, and present data and analysis.

4.1. Objects of Analysis

Five open source web applications written in PHP are used as objects of analysis for this study; the applications were obtained from different source code repository such as SourceForge. The applications are described below.

osCommerce [7] (open source Commerce) is a web based store-management and shopping cart application. FAQForge [8] is a web application used to create FAQ (frequently asked questions) documents, manuals, and HOWTOs. Three versions are used for both osCommerce and FAQForge.

phpScheduleIt [29] is a web application that attempts to solve the problem of scheduling and managing resource utilization. It provides a permission-based calendar that allows users to self-register and to reserve resources and tools to manage those reservations. Some typical applications are scheduling a conference room, equipment, or work shift etc. Four versions of this object program are used for the experiment.

Mambo [30] is a content management system that can be used for everything from simple websites to complex corporate applications. It is used worldwide to power government portals, corporate intranets and extranets, ecommerce sites, and nonprofit outreach, school, church, and community sites. Seven versions of this object program are used for the experiment.

The last application we used is Mantis [31], a web-based bug tracking system. Mantis supports multiple DBMS, such as the MySQL, MS SQL, and PostgreSQL databases, and works on multiple platforms. It provides various bug tracking functionalities, including change log support, source control integration, and time tracking. Due to its complicated functionalities, Mantis is the largest one among the applications with which the experiment was done. (The latest version is over 200KLOC.) For this object program, eight versions were used to conduct the experiment.

All these object programs are real, non-trivial web applications that have been utilized by a large number of users.

Table 4 shows the list of the objects, their associated “Version,” “Lines of Code,” and “No. of Files.” The lines of code count in this table include both PHP code and HTML markups. PDG construction includes HTML markups in the form of PHP echo statement for change analysis.

Table 4. Objects of Analysis

Application	Version	Lines of Code	No. 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
	2.2MS2-060817	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
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
phpScheduleIt	1.0.0	35045	90
	1.1.0	59753	143
	1.2.0	63138	178
	1.2.12	72396	192

4.2. Variables and Measures

4.2.1. Independent Variables

The empirical study manipulated one independent variable, test input generation technique. One control technique and one heuristic technique are considered.

The control technique (the original PARTE approach) generates executable test cases without utilizing reusable input constraint values. This technique serves as an experimental control. The heuristic technique generates executable test cases utilizing reusable input constraint values by analyzing program dependence graphs and definitions/uses information for the reusable variables explained in earlier chapters.

4.2.2. Dependent Variable and Measures

The dependent variable is the number of reusable input values identified by the technique and the percentage of reusable input values over the total number of input values.

4.3. Experiment Setup

The experiment was setup using a virtual machine on multiple hosts. The collected data is not associated with time because different virtual machine hosts with different performance capabilities were used. The operating system for the virtual machine was Ubuntu Linux version 10.10. The server ran Apache as its HTTP server and MySQL as its database backend. PHP version 5.2.13 and Zend engine v2.2.0 were used.

The tool was written in Java, and the Oracle/Sun JRE and JDK version 6 were used as the development and execution platforms. PHC version 0.2.0.3 was used to parse the PHP files. Perl and Bash scripts were used to control the modules and to pass data throughout the tool chain. A tool chain automation script from PARTE is modified to accommodate the reuse module in the existing framework.

The total number of input values required for the set of new paths and the number of reusable input values were collected to calculate the percentage of reusable input values. The percentage of Reuse is calculated over Total.

4.4. Threats to Validity

In this chapter, the internal and external threats to the validity of the study are described. Also, the approaches that are used to limit the effects of these threats are described.

4.4.1. Internal Validity

The inferences that are made about the efficiency of the approach could have been affected by the following factors. (1) The methodology is dependent on program dependence graphs generated in the earlier phase of PARTE. The result is directly related to the proper generation of the PDGs. To avoid any issue with the PDG generator, the tool has been thoroughly examined and any existing inconsistencies are removed. (2) Partial matching for statement values of a particular variable sometimes becomes tricky for complex expression. However, different scenarios have been handled to avoid any discrepancy with partial matching.

4.4.2. External Validity

Several open source web applications are used for the study, so these programs are not representative of the applications used in practice. However, this threat is minimized by using multiple non-trivial sized web applications that have been utilized by many people.

Initially only two object programs were used to conduct empirical analysis. But, to address this threat more rigorously, additional experiments have been performed by adding three more in the object program pool.

CHAPTER 5. DATA AND ANALYSIS

In this chapter, the results of the study and data analyses are presented considering the research question. Further implications of the data and results are discussed in Chapter 6. The research question considers whether the approach can be efficient in reducing efforts to generate new test cases during regression testing. To answer this question, the results collected with and without using reusable input values are compared. Three sets of data are gathered: (1) the total number of input values that are required to execute new test paths, (2) the number of reusable input values, and (3) the percentage for the number of reusable input values over the number of all input values.

Before discussing the result of input values, an overview of how many test paths are required to test modified versions of the applications is provided in Table 5. For the latest version of FAQForge, a total of 73 test paths are required, and in case of osCommerce, a total of 2409 test paths are required when code changes are not considered. Although, the number of test paths are significantly reduced by considering only the areas affected by code changes, still a large number of test paths are required to regression test (e.g., for the version 2.2MS2 of osCommerce, 1719 paths are needed.)

Table 6 summarizes the data gathered from running my technique on FAQForge and osCommerce. The table lists, for each application, the "Version Pair" (two versions of the application analyzed), "Total Input Values" (the total number of input values are required for executable new test paths), and "Reusable Input Values" (the number of reusable input values). Table 7 shows the percentage of input values that can be reused over the control technique. The data is shown in version pair because regression testing starts from the second release of the application.

Table 5. Total Number of Paths and Regression Paths

Application	Version	Total Number of Paths	Regression Paths (Number of Paths for Code Changes)
FAQForge	1.3.1	73	5
	1.3.2	73	19
osCommerce	2.2MS2	2403	1719
	2.2MS2-060817	2409	58
Mambo	4.5.6	1357	65
	4.6.1	1388	527
	4.6.2	1416	236
	4.6.3	1409	92
	4.6.4	1444	114
	4.6.5	1444	20
Mantis	1.1.7	3482	221
	1.1.8	3482	185
	1.2.0	4345	2802
	1.2.1	4373	166
	1.2.2	4389	106
	1.2.3	4403	103
	1.2.4	4419	199
phpScheduleIt	1.1.0	481	338
	1.2.0	518	314
	1.2.12	529	154

5.1. Results for FAQForge

For the version pair 1.3.0 and 1.3.1 of FAQForge, the total number of required input variables is 25. Among 25 total input values, 19 input values are reusable. For FAQForge, the size of the application is relatively small (i.e. 1671 lines of code and 18 files for the latest version) compared to other object programs of the experiment. The changes between versions were also very small; therefore the number of input values required for the new paths are relatively small, and this result is not surprising.

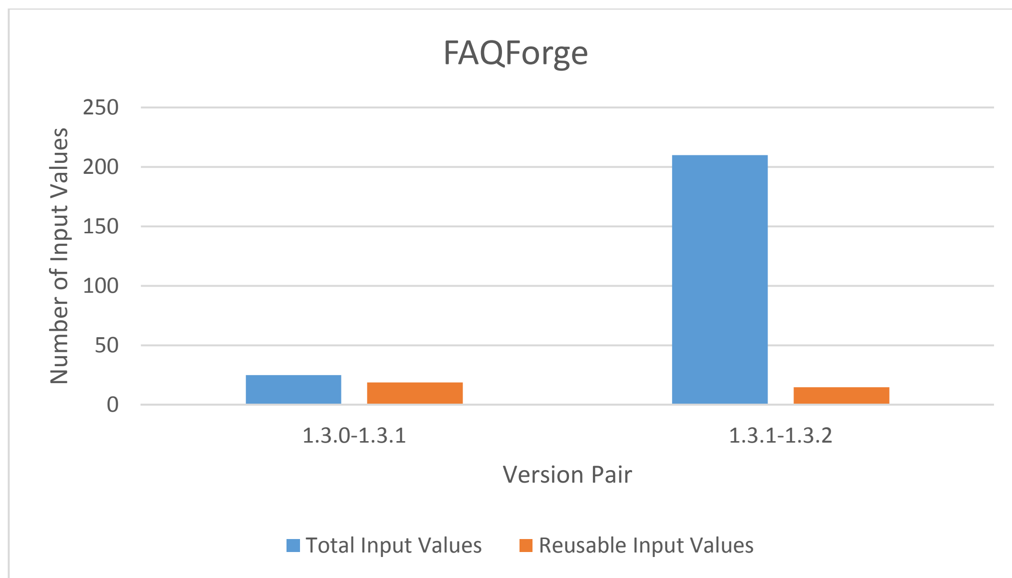

Figure 4. FAQForge Input Values Comparison

By inspecting the files in FAQForge, it is found that only one of the files in version 1.3.1 is changed from version 1.3.0. The changed file is a library file that contained functions included by the main index file. During path generation, no library files were directly analyzed. Instead, the path generator analyzed files that the user would execute directly.

For the second pair of FAQForge (versions 1.3.1 and 1.3.2), unlike the first version pair, this pair had many changes in the source files and produced a high number of paths as well as a high number of input values. The total numbers of input values was 210, and among those only

15 input values were reusable. Manual inspection of the source files revealed that 12 files changed for this version pair, which explains the higher number of input values than the first version pair.

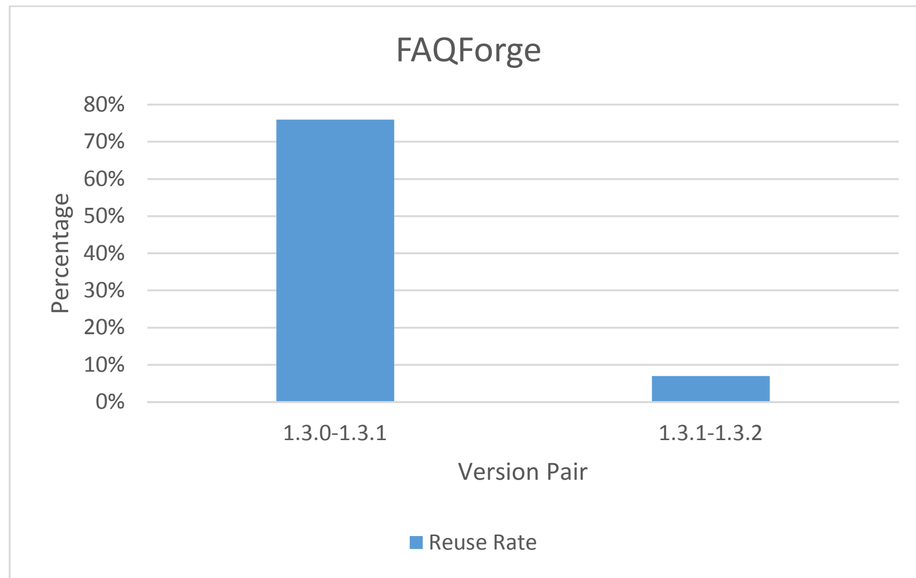

Figure 5. FAQForge Reuse Rates

To understand the low number of reusability, differences of the two version of PDG are reviewed. It is found that most of the changes in the source files are simple output statements that have no data dependencies. A common example in PHP would be to echo or print static HTML statements. If the static text changed, the statement was marked as a difference. For these files, the PHP variables were not affected by changes, and as a result, numbers of reusable variables were significantly low.

As shown in Table 7, the technique required a relatively small number of test input values compared to the control technique. The technique was able to reuse input values by 76% and 7% for versions 1.3.1 and 1.3.2, respectively.

Table 6. The Number of Input Values (Total vs. Reusable)

Application	Version Pair	Total Input Values	Reusable Input
FAQForge	1.3.0 & 1.3.1	25	19
	1.3.1 & 1.3.2	210	15
osCommerce	2.2MS1 & 2.2MS2	4392	702
	2.2MS2 & 2.2MS2-060817	813	219
Mambo	4.5.5 & 4.5.6	490	294
	4.5.6 & 4.6.1	4446	67
	4.6.1 & 4.6.2	2075	536
	4.6.2 & 4.6.3	1236	250
	4.6.3 & 4.6.4	1567	191
	4.6.4 & 4.6.5	324	57
Mantis	1.1.6 & 1.1.7	1524	708
	1.1.7 & 1.1.8	1238	662
	1.1.8 & 1.2.0	20425	195
	1.2.0 & 1.2.1	1772	558
	1.2.1 & 1.2.2	1042	296
	1.2.2 & 1.2.3	890	267
	1.2.3 & 1.2.4	2643	263
phpScheduleIt	1.0.0 & 1.1.0	2389	280
	1.1.0 & 1.2.0	3877	472
	1.2.0 & 1.2.12	2339	861

Table 7. Input Values Reuse Rates

Application	Version Pair	Reuse rate
FAQForge	1.3.0 & 1.3.1	76%
	1.3.1 & 1.3.2	7%
osCommerce	2.2MS1 & 2.2MS2	16%
	2.2MS2 & 2.2MS2-060817	27%
Mambo	4.5.5 & 4.5.6	60%
	4.5.6 & 4.6.1	2%
	4.6.1 & 4.6.2	26%
	4.6.2 & 4.6.3	20%
	4.6.3 & 4.6.4	12%
	4.6.4 & 4.6.5	18%
Mantis	1.1.6 & 1.1.7	46%
	1.1.7 & 1.1.8	53%
	1.1.8 & 1.2.0	1%
	1.2.0 & 1.2.1	31%
	1.2.1 & 1.2.2	28%
	1.2.2 & 1.2.3	30%
	1.2.3 & 1.2.4	10%
phpScheduleIt	1.0.0 & 1.1.0	12%
	1.1.0 & 1.2.0	12%
	1.2.0 & 1.2.12	37%

5.2. Results for osCommerce

For osCommerce, the number of source files is large compared to FAQForge. From the manual inspection of the source for the first pair (versions 2.2MS1 and 2.2MS2), it was found that 279 of the 506 files had changed. The modified files were in every module of the application, and the files with the largest differences were the library files that were included in the executable files.

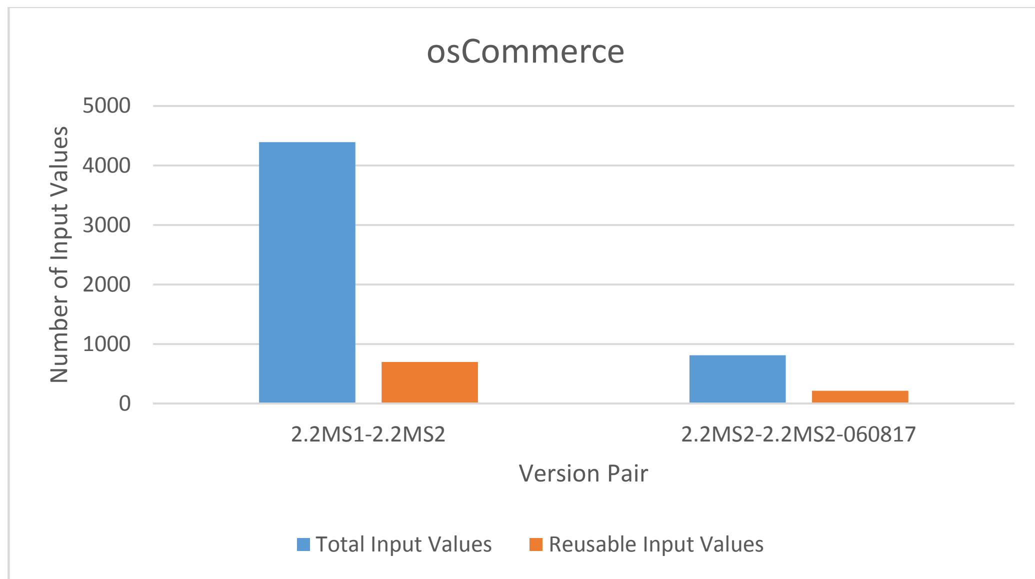

Figure 6. osCommerce Input Values Comparison

For the first version pair of osCommerce, 4392 input values need to be solved to execute all the regression test paths. Among these values, 702 were reusable for regression testing the new version of source code.

For the second version pair (2.2MS2-2.2MS2-060817), the manual inspection showed that 105 of the 502 files had changed. The number of changed files was smaller than the first version pair. For this pair, a total of 813 input values needed to be solved to execute all the regression test paths. Among these values, 219 were reusable for regression testing of the new version of source code.

Considering the number of files that were changed for both versions (279 of 506 for version 2.2MS2 and 105 of 502 for version 2.2MS2-060817), the number of required input values was relatively small (4392 for version 2.2MS2 and 813 for version 2.2MS2-060817). The reason for this is that there were numerous changes in statements that contained no variables.

Similar to the results for FAQForge, the heuristic technique required a relatively small number of test input values compared to the control technique (Table 7). The technique was able to reuse input values by 16% and 27% for versions 2.2MS2 and 2.2MS2-060817, respectively.

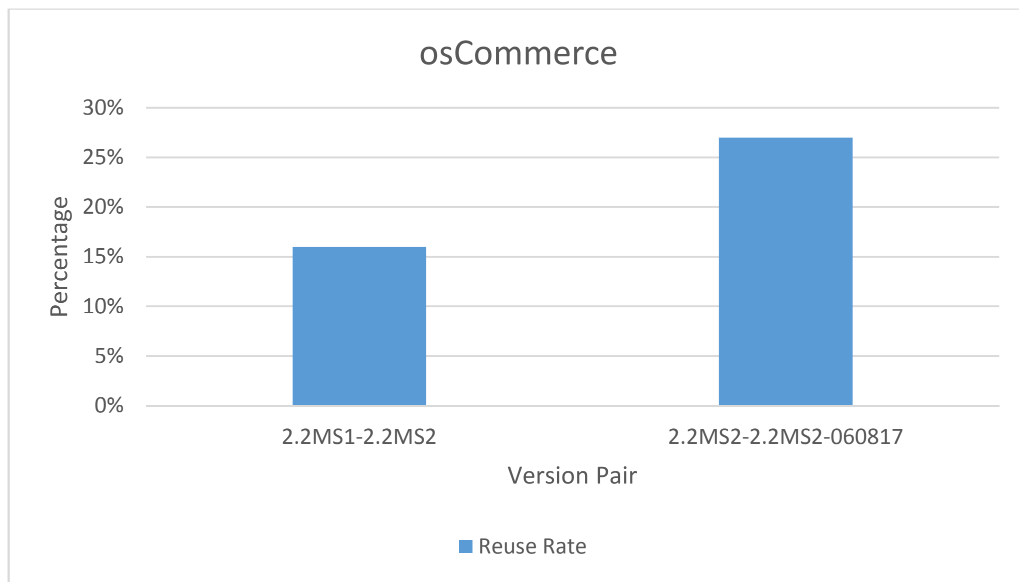

Figure 7. osCommerce Reuse Rates

The result also indicates an important fact that the heuristic technique of this research is more efficient than the control technique if applied on two version pairs with few changes between them. That means, if new version of an object program contains small patch or bug fix from the previous version, the regression test would benefit more using constraint reuse technique.

5.3. Results for Mambo

The experiment is run on a total of six version pairs and a varying result for different version pairs of Mambo is found. Among those six version pairs, the version pair 4.5.6 - 4.6.1 contains the most number of changes. The release note of version 4.6.1 also mentions that version 4.6.1 is a major update with lots of feature changes and bug fixes. The high amount of changes reduced the reusability significantly for the version pair. Only 2% of the variables are reusable to the new version.

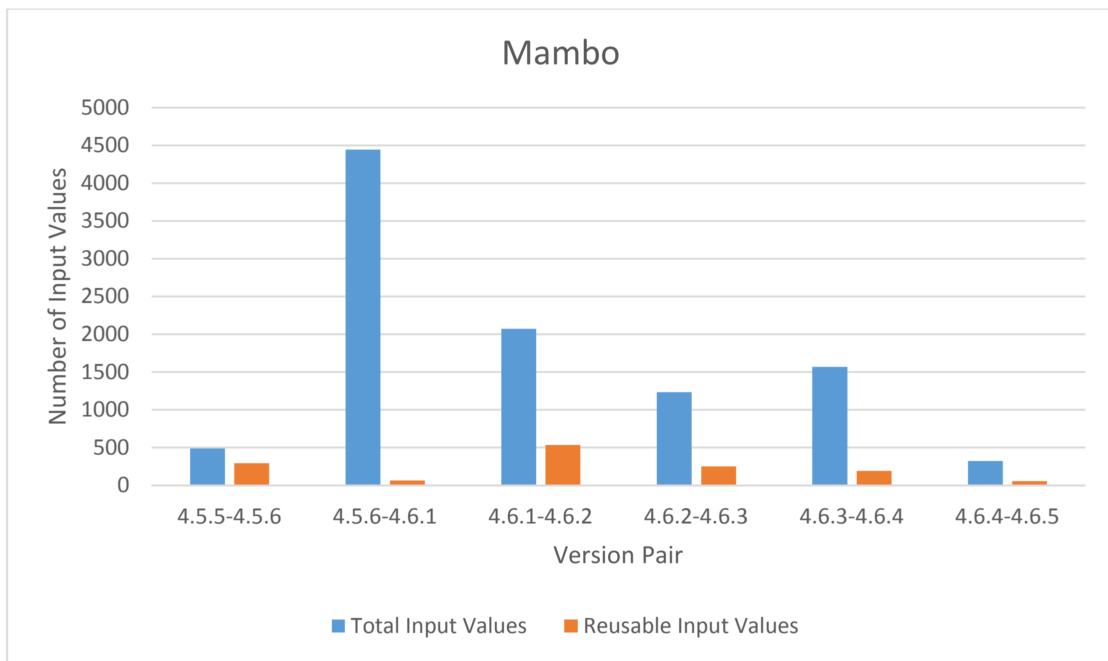

Figure 8. Mambo Input Values Comparison

Other version pairs also contained a lot of changes from previous version. In fact, none of the versions can actually be treated as a small patch update or small bug fixes.

The first version pair 4.5.5 & 4.5.6 has the most efficient reusability count with 60% of the input variable values reusable for the new version. The changes were lower compared to other version pairs which justifies the high number of reusability.

The rest of the version pairs produced on average 19% reusable input variables which indicate a good amount of savings for regression test. The results again prove the fact that version pairs with small patch update or bug fix can gain efficiency by using this heuristic technique.

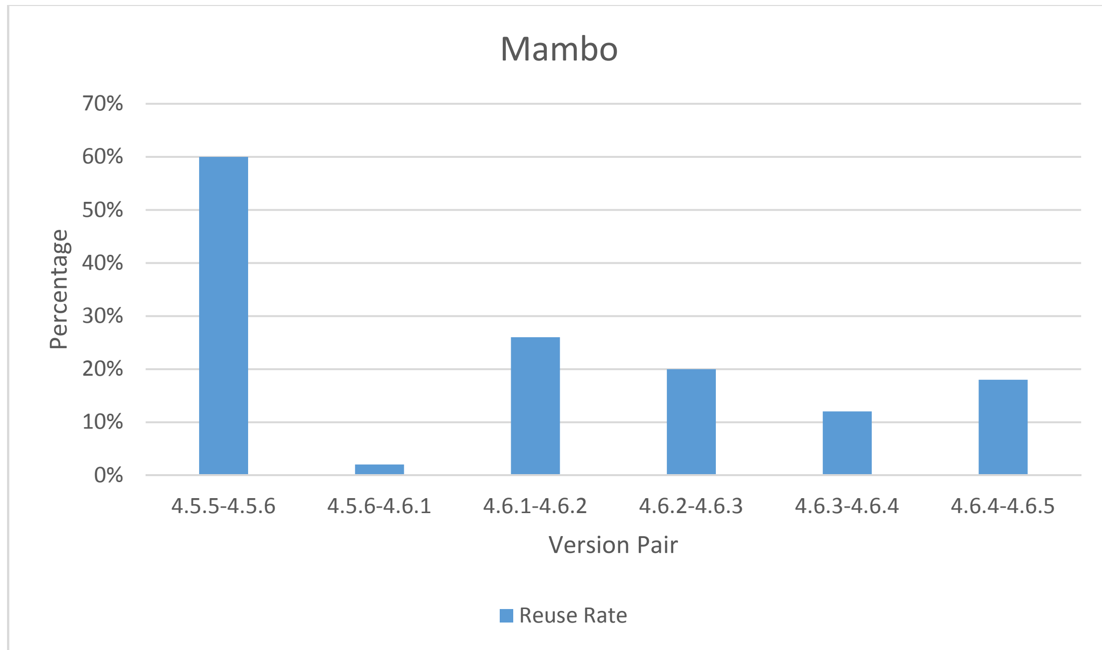

Figure 9. Mambo Reuse Rates

5.4. Results for Mantis

The highest number of version pairs is used for Mantis in this experiment. Seven version pairs of Mantis provided a good picture of benefit from the reusability data. Most of the version pairs showed significant reusability.

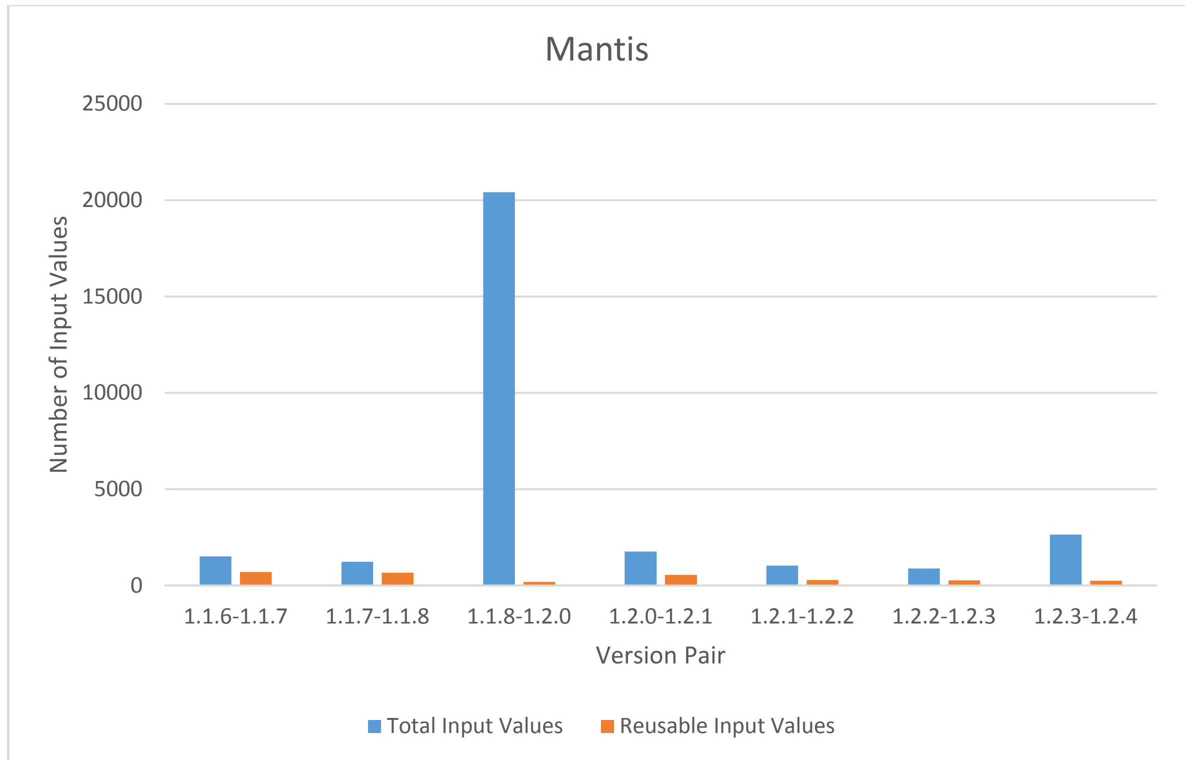

Figure 10. Mantis Input Values Comparison

Version pair 1.1.8 & 1.2.0 showed very low percentage of reusability. Only 1% of the input variable values were reusable for regression test path generation of version 1.2.0. Analysis of the source code for the two versions revealed that a large number of files contained changes from version 1.1.8 to version 1.2.0. Also, the release note of version 1.2.0 mentioned a lot of feature changes and bug fixes. Version 1.2.0 was a major update for Mantis. This resulted paths which are a lot different from the previous version as well as low number of reusable variables.

Other version pairs showed on average 33% reusability, which indicates a significant effort reduction in constraint resolution as well as regression testing. Two of the version pairs 1.1.6 - 1.1.7 and 1.1.7 – 1.1.8 had nearly 50% of the input variables reusable from the previous version pairs.

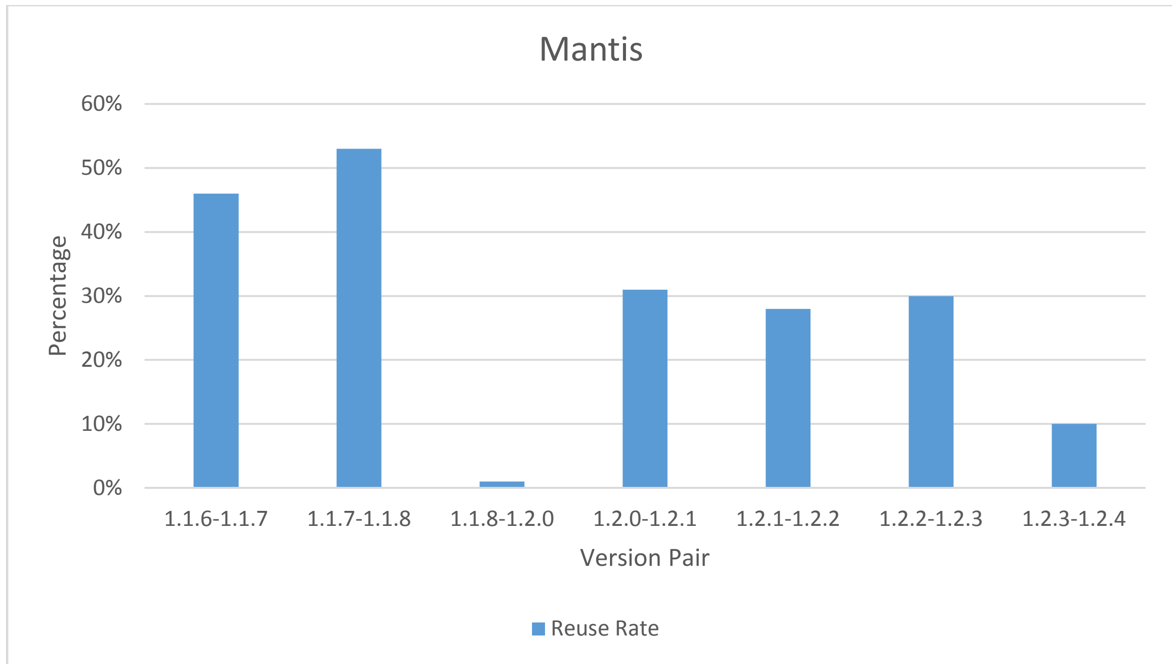

Figure 11. Mantis Reuse Rates

The reusability data also revealed similar observation of other object programs. Version pairs with small patch update or bug fix can gain efficiency by using the heuristic technique.

5.5. Results for phpScheduleIt

Three version pairs are used for phpScheduleIt in this experiment. The results showed lower reusability for the object program. First two version pairs showed an average 12% reusability while the last version pair showed a good reusability result of 37%.

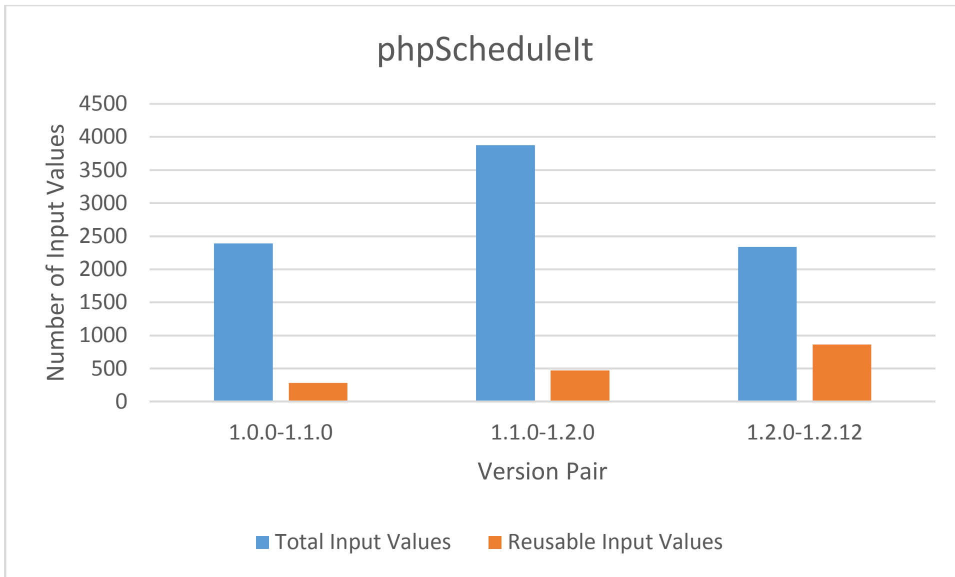

Figure 12. phpScheduleIt Input Values Comparison

To investigate the result, the source code of the first two version pairs is reviewed. It is found that update from 1.0.0 to 1.1.0 and 1.1.0 to 1.2.0 involved large functionality change and bug fixes. Generated paths for those version pairs were mostly incompatible. As a result, a small number of variables are found to be eligible for reuse.

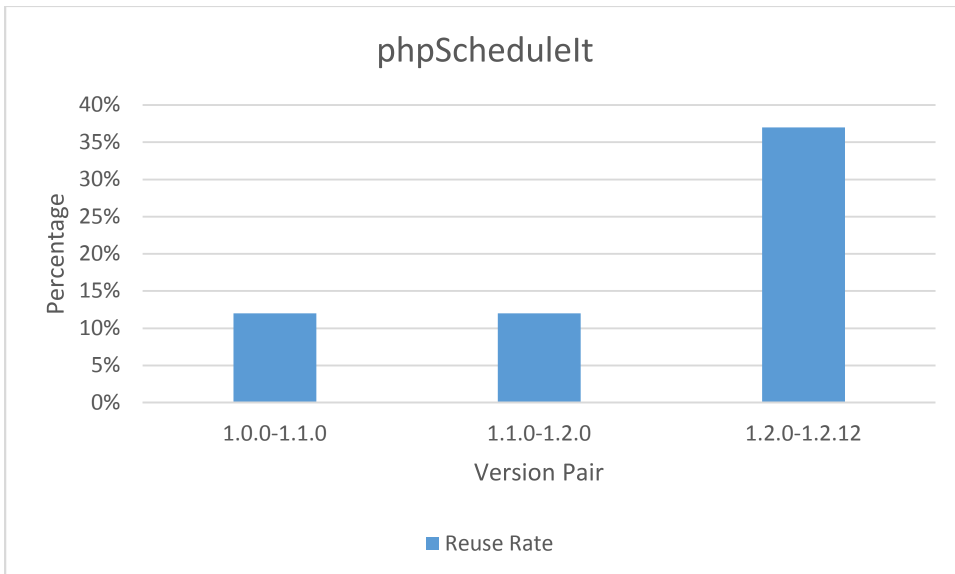

Figure 13. phpScheduleIt Reuse Rates

The final version pair (1.2.0 – 1.2.12) has good reusability percentage (i.e. 37%). When the source code for the two versions is reviewed, only a few source code files are found to be changed. The release note of version 1.2.12 also confirmed that the update from 1.2.0 to 1.2.12 was for minor patching and few bug fixes.

Similar to other object programs, phpScheduleIt indicates the fact that version pairs with small patch update or bug fix can gain more efficiency by using the heuristic technique.

CHAPTER 6. DISCUSSION

The experiment results strongly suggest that input variable value reuse for regression test cases can save a significant amount of time and effort during regression testing. As observed from the data analysis, the reuse rates of input values with the approach over the control technique (original PARTE approach) are significantly high for most of the version pairs of the applications. For some of the version pairs, the reuse rates were exceptionally high. The first version pair (1.3.0 - 1.3.1) for FAQForge yielded a 76% reuse rate, and the version pair (4.5.5 - 4.5.6) for Mambo yielded 60%. Also, two version pairs of Mantis (1.1.6 - 1.1.7 and 1.1.7 - 1.1.8) showed reuse rates of 46% and 53%.

As PARTE requires actual input values to create executable test cases, reusing some of the input values can reduce regression testing time significantly, and this means a short turn-around time in releasing patches can be achieved. For instance, in the case of osCommerce, the number of test paths generated using program slice information was 1719 for version 2.2MS2. For those test paths, it was learned from previous study [1] that a large number of constraints were not solvable by automatic resolvers. Instead, many input values had to be resolved manually, and it was a time consuming process. The approach that facilitates constraint reuse allowed a significant reduction of efforts during the constraint resolution process.

It was also observed that there is a relationship of reuse rate with the number of changes in the version pair. For FAQForge, the approach identified more reusable input values from the first version pair than the second version pair. 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. It was revealed by inspection that the number of changed

files was fewer in the second version pair than the first version pair. The data collected from the rest of the object programs also showed the similar behavior.

The type of changes between the versions can impact the reuse rate. Experiment results on Mambo, Mantis and phpScheduleIt demonstrate this impact. Versions 4.5.5 and 4.5.6 of Mambo has a reuse rate of 60%. A manual review of the change history showed that there were only minor bug fixes between the two versions. Whereas, versions 4.5.6 and 4.6.1 had significant code changes that lead to a drop in reuse rate to 2%. In case of Mantis, a total of 149 changes were found between versions 1.2.0 and 1.1.8. These changes include bug fixes and introduction of new features. This produced new test paths, and dropped constraint reuse rate to 1%. The changes between versions 1.1.6 and 1.1.7 were due to 18 bug fixes. No new source code or resource files were added. Hence the constraint reuse rate was relatively higher at 46%. In phpScheduleIt, the changes from version 1.2.0 to 1.2.12 were related to language support and several minor bug fixes. Whereas, among versions 1.0.0, 1.1.0 and 1.2.0 there were new features along with bug fixes. For example, 1.1.0 introduced support for multiple day reservations and 1.2.0 allows additional resources to be added to an existing reservation.

These observations clearly state the fact that object program 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 significant amount.

To our knowledge, this study is 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 provide an insight about how reusable constraint values can be utilized during the testing and regression testing process.

CHAPTER 7. CONCLUSION

In this thesis, a technique is presented that identifies reusable constraint values for regression test cases by analyzing definitions and uses of the variables for two consecutive versions. To evaluate the approach, an experiment was conducted using five open source web applications (small and large), and the results showed a large number of constraints input values which can be reused from the previous version's test cases. Thus it can reduce a significant amount of effort for resolving constraint values for those variables when new versions of the application is tested.

Further, the constraints and actual values for variables can be reusable across several versions as long as the definitions and uses relationships of the variables hold across versions. This means that greater savings can be expected as the applications evolve over time.

While the approach can reduce the amount of time needed to apply regression testing for patched web application software, it can also reduce the time and effort as well as improve testing effectiveness when the major releases are tested because new test cases and other associated artifacts accumulate over time. Also, by combining the approach with automatic resolution using constraint solvers (for instance, feeding initial input values to the solvers by analyzing existing executable test cases), regression testing processes can be accelerated even faster.

The results of the studies suggest some future work. Five widely used open source web applications with three or more versions were used to evaluate this approach. While the experiment results are promising, the approach observed only version pairs and did not observe the effect of reuse propagation with groups of version pairs. Also, for some object programs the experiment was run on small number of versions. This means that the experiment results

obtained in this study do not sufficiently capture possible benefits and factors related to the long-term utilization of the technique. Certainly, it is believed that the approach would provide greater benefits when applied over a long period 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).

Additional studies can be performed that apply the approach to a wider population (e.g., larger open source applications and industrial-size applications) with different testing processes (e.g., constraints imposed by an industry's regression testing practice).

Further, in this work, existing test cases were utilized to extract reusable variables, their constraints, and their input values, but the actual existing test cases were not used for testing the new version of the program (in this context, test case selection). However, by reusing the existing test cases when we test the modified program, additional savings can be achieved. Thus, there is a future plan to investigate test case selection approaches that choose test cases that exercise the modified areas of code to help reduce the cost of generating new tests.

REFERENCES

[1] A. Marback, H. Do, and N. Ehresmann, “An effective regression testing approach for php web applications,” *International Conference on Software Testing, Verification and Validation*, Apr. 2012, pp. 221–230.

[2] P. Hooimeijer and W. Weimer, “Solving string constraints lazily,” *IEEE and ACM International Conference on Automated Software Engineering*, Sep. 2010, pp. 377–386.

[3] A. Kiezun, P. Guo, K. Jayaraman, and M. Ernst, “Automatic creation of SQL injection and cross-site scripting attacks,” *International Conference on Software Engineering*, May 2009, pp. 199–209.

[4] phc, “phc - the open source php compiler,” http://www.phpcompiler.org/.

[5] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, and M. Ernst, “Hampi: a solver for string constraints,” *International Conference on Software Testing and Analysis*, Jul. 2009, pp. 105–115.

[6] Choco, “Choco solver website,” http://www.emn.fr/z-info/choco-solver/.

[7] osCommerce, “osCommerce website,” http://www.oscommerce.com/.

[8] FAQforge, “FAQforge website,” http://sourceforge.net/projects/faqforge/.

[9] S. Elbaum, A.G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” *IEEE Transactions on Software Engineering*, Feb. 2002, vol. 28, no. 2, pp. 159–182.

[10] G. Rothermel and M.J. Harrold, “A safe, efficient regression test selection technique,” *ACM Transactions on Software Engineering and Methodology*, Apr. 1997, vol. 6, no. 2, pp. 173–210.

[11] A. Walcott, M.L. Soffa, G.M. Kapfhammer, and R.S. Roos, “Time-aware test suite prioritization,” *International Conference on Software Testing and Analysis*, Jul. 2006, pp. 1–12.

[12] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," *International Conference on Software Testing and Analysis*, Jul. 2007, pp. 140–150.

[13] T. Apiwattanapong, R. Santelices, P.K. Chittimalli, A. Orso, and M.J. Harrold, "MATRIX: Maintenance-oriented testing requirements identifier and examiner," *Academic and Industrial Conference - Practice And Research Techniques*, Aug. 2006, pp. 137–146.

[14] R. Santelices and M.J. Harrold, "Applying aggressive propagation-based strategies for testing changes," *International Conference on Software Testing, Verification and Validation*, Apr. 2011, pp. 11–20.

[15] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," *ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2010, pp. 257–266.

[16] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," *International Symposium on Software Reliability Engineering*, Nov. 2011, pp. 150–159.

[17] K. Taneja, T. Xie, N. Tillmann, and J. Halleux, "eXpress: Guided path exploration for efficient regression test generation," *International Conference on Software Testing and Analysis*, Jul. 2011, pp. 1–11.

[18] Y. Chen, R. Probert, and H. Ural, "Model-based regression test suite generation using dependence analysis," *International Workshop on Advances in Model-Based Testing*, Jul. 2007, pp. 54–62.

[19] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," *International Symposium on Software Testing and Analysis*, Jul. 2008, pp. 249–260.

[20] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, “Practical fault localization for dynamic web applications,” *International Conference on Software Engineering*, May. 2010, pp. 265–274.

[21] F. Ricca and P. Tonella, “Analysis and testing of web applications,” *International Conference on Software Engineering*, May. 2001, pp. 25–34.

[22] Y. Deng, P. Frankl, and J. Wang, “Testing web database applications,” *ACM SIGSOFT Software Engineering Notes*, 2004, vol. 29, no. 5, pp. 1–10.

[23] W. Halfond, S. Anand, and A. Orso, “Precise interface identification to improve testing and analysis of web applications,” *International Conference on Software Testing and Analysis*, Jul. 2009, pp. 285–296.

[24] K. Dobolyi and W. Weimer, “Harnessing web-based application similarities to aid in regression testing,” *International Symposium on Software Reliability Engineering*, Nov. 2009, pp. 71–80.

[25] S. Elbaum, S. Karre, and G. Rothermel, “Improving web application testing with user session data,” *International Conference on Software Engineering*, May. 2003, pp. 49–59.

[26] N. Klarlund, “Mona Fido: The logic-automaton connection in practice,” *Conference on Computer Science Logic*, Aug. 1997, pp. 311–326.

[27] M. Emmi, R. Majumdar, and K. Sen, “Dynamic test input generation for database applications,” *International Symposium on Software Testing and Analysis*, Jul. 2007, pp. 151–162.

[28] G. Wassermann and Z. Su, “Sound and precise analysis of web applications for injection vulnerabilities,” *ACM SIGPLAN conference on Programming language design and implementation*, Jun. 2007, pp. 32–41.

[29] phpScheduleIt, “phpScheduleIt website,” http://www.php.brickhost.com/.

[30] Mambo, “Mambo website,” http://www.mamboserver.com/.

[31] Mantis, “Mantis website,” http://www.mantisbt.org/.

[32] N. Alshahwan and M. Harman, “Augmenting test suites effectiveness by increasing output diversity,” *International Conference on Software Engineering*, Jun. 2012, pp. 1345–1348.

[33] K. Rubinov and J. Wuttke, “Augmenting test suites automatically,” *International Conference on Software Engineering*, Jun. 2012, pp. 1433–1434.