Regression Testing for Web Applications Using Reusable Constraint Values

Md. Hossain *
* Microsoft
md.i.hossain@ndsu.edu

Hyunsook Do †
† North Dakota State University
Computer Science
hyunsook.do@ndsu.edu

Ravi Eda ‡
‡ Microsoft
North Dakota State University
ravi.eda@ndsu.edu

***Abstract*—Companies that provide web applications need to perform frequent regression testing because companies often encounter various security attacks and frequent feature update demands from users. Typically, such applications require regression testing processes that require minimal test effort because they have already been deployed and used in the field. In our previous work, we presented an efficient regression testing approach that allows us to focus on the areas of code that have been changed and to regression test them to address this problem. While our experiment results showed that this approach can be efficient in saving the cost of regression testing by reducing the number of test paths necessary for the modified program, we also learned that resolving input constraints requires a lot of effort. In this paper, to accommodate further savings with regression testing, we present a technique that identifies reusable constraint values for regression test cases. The technique finds variables where input values from the previous version can be reused to execute the regression test path for the new version. Our empirical results show that a large number of variable constraints can be reused from the previous version's test cases, thus we can reduce a great deal of effort for resolving constraint values for those variables when we test a new version of the application.**

***Keywords*-Regression testing, test case generation, reusable constraint values, PHP web applications**

I. Introduction

The use of web applications has grown rapidly over the past decade, and a large number of companies rely 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 these needs arise, companies need to fix security problems or upgrade the application with new features. Thus, such applications undergo numerous patch releases which require frequent regression testing processes that require minimal test effort to release patches instead of applying regression testing to the entire product.

In our previous work, we introduced a new regression testing approach PHP Analysis and Regression Testing Engine (PARTE) [1]) that allows us to focus on the areas of code that have been changed and to regression test them. In particular, we developed a technique that generates test cases using program slices and their inputs that require constraint resolution. Our experiment results with two open web applications indicated that our approach is efficient in reducing the cost of regression testing by reducing the number of test cases to exercise. We also learned that resolving input constraints was a tedious 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 we can identify input constraints and their resolved values from the previous version's test cases that are applicable to the current version, we can expect greater savings. A recent work by Visser et al. [4] demonstrates the benefits of persisting resolved constraint values so that these values can be reused not just for the application under test but other applications too.

In this paper, we propose a technique 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 application, we determine whether the same constraints for the variable can be used. By doing this, we can avoid unnecessary effort during regression testing; this approach helps us avoid collecting constraints for the reusable variables; it also reduces the number of numeric and string inputs that need to be resolved.

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

The rest of the paper is organized as follows. In the next section, we describe our overall methodology, including a brief description of PARTE and the technical details about the proposed approach. Sections III and IV present our experiment design, results, and analysis. Section V discusses the results and their implications. Section VI describes related

work relevant to web applications and regression testing, and Section VII presents conclusions and future work.

II. Methodology

As presented in our previous paper [1], we implemented PARTE to generate new regression test cases by analyzing the code elements impacted by code changes. To facilitate the approach proposed in this paper, we extend PARTE as shown in Figure 1. The dark gray area is the new addition to PARTE. To provide an overview for our approach, we summarize PARTE's main components and then explain the components related to the constraint resolution reuse.

In Figure 1, the boxes depict the main activities; the ovals appeared outside the boxes are external tools that PARTE utilizes (PHC, Choco, and Hampi). PARTE has three main components: (1) Preprocessing: Before we perform the impact analysis on PHP web applications, a preprocessing step is required to handle dynamic aspects of PHP web applications and to preserve variable names across versions because we use a PHP compiler, PHC [5] and because PHC 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, we use two constraint solvers, Hampi [6] and Choco [7]. Choco was selected for its ease of use and Hampi for it effective string variable solving capability.

To collect reusable constraints and input values from the previous version, we require the following steps. (To apply this approach, we already have test paths and executable test cases that have been used for testing the previous version. In Figure 1, the database for version v0 at the bottom of the figure contains these two sets of information.) (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, the test paths required for the new version are generated. (2) Two sets of 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.

We describe these two steps in detail in the following subsections, including an algorithm and an example, that show how our approach works.

A. Collecting Reusable Constraint Values

Without considering the use of reusable constraint input values, a typical way to generate executable test cases utilizing PARTE is as follows. Once all necessary preprocessing activities are done, 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, we need to assign actual input values for the input variables. To do so, the constraint collector gathers constraints for these input variables, and then, the constraint resolution tool generates the input values that satisfy the constraints. (We need to resolve input values manually if the tool cannot resolve them.)

However, as we briefly mentioned in Section I, 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 code changes could be large, thus the number of inputs could large. To address this issue, we implemented a technique that collects reusable input constraint values from the previous version's

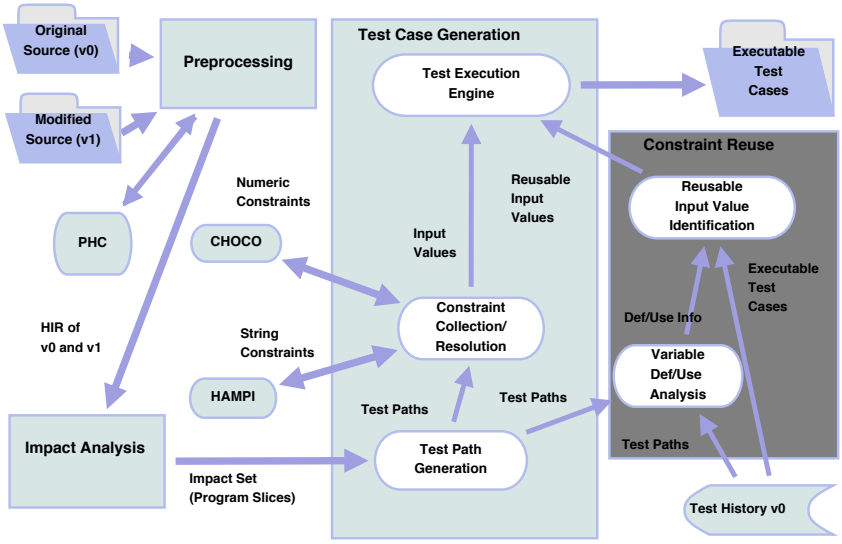

Fig. 1: Overview of Our Approach

Algorithm 1 FindReusableInputValues

```
1: Inputs: oldPath, newPath, oldPDG, newPDG
2: Output: reusablevariables
3: newPathVarDefUseMap : mapping variable to DefUse statement for the new path
4: for n ← 0, n < newPath.size(), n++ do
5:     currentBlock ← newPDG.getBlock(newPath.getBlockID(n))
6:     if currentBlock.HasDefOrUse() then
7:         UpdateDUM(currentBlock, newPathVarDefUseMap)
8:     end if
9: end for
10: oldPathVarDefUseMap : mapping variable to DefUse statement for the old path
11: for n ← 0, n < oldPath.size(), n++ do
12:     currentBlock ← oldPDG.getBlock(oldPath.getBlockID(n))
13:     if currentBlock.HasDefOrUse() then
14:         UpdateDUM(currentBlock, oldPathVarDefUseMap)
15:     end if
16: end for
17: reuseVars : list of variables whose input values can be reused
18: for n ← 0, n < newPathVarDefUseMap.getVars.size(), n++ do
19:     curVar ← newPathVarDefUseMap.getVars()[n]
20:     if oldPathVarDefUseMap.getVars().contains(curVar) then
21:         isSim ← ChkSim(curVar, oldPathVarDefUseMap, newPathVarDefUseMap)
22:         if isSim then
23:             reuseVars.Add(curVar)
24:         end if
25:     end if
26: end for
27: return reuseVars
```

Algorithm 2 ChkSim

```
1: Inputs: curVar, oldPathVarDefUseMap, newPathVarDefUseMap
2: Output: true or false
3: oldPathS ← oldPathVarDefUseMap.getStmts(curVar);
4: newPathS ← newPathVarDefUseMap.getStmts(curVar);
5: for n ← 0, n < newPathS.size(), n++ do
6:     if !CompMatch(newPathS[n],oldPathS[n]) then
7:         if !ParMatch(newPathS[n], oldPathS[n]) then
8:             return false
9:         end if
10:    end if
11:    if HasVDepend(newPathS[n], oldPathS[n]) then
12:        isResolved ← ResolveVarDependency(newPathS[n], oldPathS[n]));
13:        if !isResolved then
14:            return false
15:        end if
16:    end if
17: end for
18: return true
```

executable test cases.

As shown in Figure 1, our technique consists of two processes (right, the 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 that were generated by the path generator and the test paths from test history database from 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 our path generator do not provide variable information, but PDGs contain definition and use information for variables. The analyzer parses these two information types 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 previous version's mapping table and executable test cases, assigning values to the reusable variables by extracting input values from the previous version's executable test cases.

The process we just explained is formally shown in the *FindReusableInputValues* and *ChkSim* algorithms (Algorithms 1 and 2). The *FindReusableInputValues* algorithm takes four inputs: old test paths, new test paths, old PDGs, and new PDGs. ("Old" indicates the previous version, and "new" indicates the current version being tested.) The algorithm produces one output: reusable variables with actual values. In line 3, a new map is declared; the map maps variable names to the list of PDG nodes that contains definitions and uses for that particular variable in 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 5). If the current node has a variable with definition and/or uses, the map is updated by calling *UpdateDUM* which changes or inserts the variable information (lines 6 and 7). A map is also defined for the old path by following the same procedure (lines 11 to 16). 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 way that we just explained.

Then, the source code statements between the two versions (old and new) are compared using a *ChkSim* (Algorithm 2) function (line 21). If the statements match, then the variable is identified as reusable and added to the reusable variables list (line 23). At first, the algorithm checks to see whether the statements are an exact match. If the statements do not match completely, then a partial match is searched in the unmatched statements. 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 changed statement contains the use of variable "a" with the same condition, we consider it a partial match.

If no partial match is found in the unmatched statements, then the variable is discarded as not reusable for new test case generation. If there is at least a partial match found between two version of statements for a variable, then the algorithm proceeds to find variable dependency. Variable dependency can occur when def-use statements for a variable match with the previous version, but one or more of these def-use statements consist of other variables. For example, one of the def-use statements for variable "a" can be "if $(a > b - 7)$." This statement shows a use of variable "a" which also involves

v0.php (original version)

```
1. $a = $_POST['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4.  $b = 5;
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";
```

v1.php (modified version)

```
1. $a = $_POST['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4.  $b = 5;
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";
```

v2.php (modified version)

```
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";
```

Fig. 2: Three Versions of the PHP Sample Program

variable "b." In this case, checking the similarity of def-use statements for variable "a" is not sufficient, because it is dependent on variable "b". That means, def-use statements for variable "b" also need to be similar between the two versions to declare reuse eligibility for variable "a".

Variable dependency for a particular variable can be resolved by checking def-use statements' similarity for all the dependent variables. If all the corresponding def-use statements match, then the variable dependency can be declared as resolved. If there is a dependency, then it must be resolved to enlist the variable as reusable.

B. An Example of Reusable Input Variable Identification

In this section, we illustrate how we identify reusable input variables using an example. Suppose we have three consecutive versions of a simple PHP program (v0.php, v1.php, and v2.php) as shown in Figure 2.

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 I. Executing this path requires constraints for the variables to be gathered and for the actual input values to be resolved. The second column of Table II shows the variable constraints for version *v1*.

TABLE I: 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 II: Constraints

v0	v1	v2
$\$a < 12$	$\$a < 12$	$\$a < 12$
$\$a - 1 > 7$	$\$a - 3 > 7$	$\$a - 3 > 7$
$\$b == 5$	$\$b == 5$	$\$b - 1 == 5$

In the tables, we added information for version *v0* as a reference. For version *v0*, there are more test paths than the one that appeared in Table I, but to simplify our explanation with the example, we only show the path that is relevant to the regression test path for version *v1*. Also, the first column of Table II shows the variable constraints for version *v0*.

In Figure 2, from version *v1* to version *v2*, we can see that statement 4 has been changed ($\$b = 5$ to $\$b = b - 1$). This case also requires one test path to be executed as shown in the fourth row of Table I. Again, constraints for the variables on that path are shown in the third column of Table II. From the second column of Table II, we can see that the constraints for variable "a" are unchanged.

To find the reusable constraints for the variables and their values, we first need to 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, "a" and "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 "a" on the paths for all three versions are statements 1 and 5. Once the definitions of "a" are collected, all uses of variable "a" are identified and stored in the map. In this case, the uses are statements 3, 5, and 7 for all three versions.

Next, for variable "a," all def-use statements are gathered from the new version's PDG and then compared with the def-use statements gathered from the old version (*v0* with *v1* and *v1* with *v2*). In this example, the def-use statements for variable "a" on the new regression path for both version pairs are 1, 3, 5, and 7. Now, we can see 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 to generate input values for version *v1*. However, for the next version pair, *v1*-*v2*, the statements that define and use variable "a" are identical, so the constraints for variable "a," including the

input value for "a," can be reused.

For variable "b," the def-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 "b" are identical, thus constraints for variable "b," including input values for "b," can be reused for regression test path execution of *v1*. For the next version pair, *v1-v2*, the def-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 "b," cannot be reused for regression test path execution of *v2*.

III. Empirical Study

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

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

To address our research question, we designed and performed an empirical study. The following subsections describe our objects of analysis, independent variables, dependent variables and measures, study setup and design, and threats to validity. Following this presentation, in Section IV, we present our data and analysis, and in Section V, we discuss practical implications of the results.

A. 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.

TABLE III: Objects of Analysis

Application	Versions	Lines of Code	No. of Files
FAQForge	3	1671	18
osCommerce	3	78892	502
phpScheduleIt	4	72396	192
Mambo	7	133475	663
Mantis	8	209345	753

osCommerce [8] (open source Commerce) is a web based, store-management and shopping cart application. *FAQForge* [9] is a web application used to create FAQ (frequently asked questions) documents, manuals, and HOWTOs. *phpScheduleIt* [10] is a web application that attempts to solve the problem of scheduling and managing resource utilization. *Mambo* [11] is a content management system that can be used for everything from simple websites to complex corporate applications. *Mantis* [12] is a web-based bug tracking system. Due to its complicated functionalities, Mantis is the largest one among the applications we used. (The latest version is over 200 KLOC.) All these programs are real, non-trivial web applications that have been utilized by a large number of users.

Table III lists, for each of our objects of analysis, data about its associated "Versions" (the number of versions of the object program), "No. of Files" (the number of files in the latest version of that program), and "Lines of Code" (the number of lines of code in the latest version of the program). The lines of code counted in this table include both PHP code and HTML markups.

B. Variables and Measures

1) Independent Variables: Our empirical study manipulated one independent variable, test input generation technique. We considered one control technique and one heuristic technique.

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 def-use information for the reusable variables explained in earlier sections.

2) Dependent Variable and Measures: Our 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.

C. Experiment Setup

We performed our study using a virtual machine on multiple hosts. Because we used several virtual machine hosts with different performance capabilities, we did not collect data associated with the execution time. 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. We used PHP version 5.2.13 and Zend engine v2.2.0.

Our 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. We modified the tool chain automation script to accommodate our module in the existing framework.

To calculate the percentage of reusable input values, we collected the total number of input values (*Total*) required for the set of new paths and the the number of reusable input values (*Reuse*). We calculated the portion of *Reuse* over *Total*.

D. Threats to Validity

In this section, we describe the internal and external threats to the validity of our study. We also describe the approaches we used to limit the effects of these threats.

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

expression. However, we have handled different scenarios to avoid any discrepancy with partial matching.

2) External Validity: We used open source web applications for our study, so these programs are not representative of the applications used in practice. However, we tried to minimize this threat by using non-trivial sized web applications with multiple versions that have been utilized by many people. To address this threat more rigorously, we need to perform additional experiments with a wide population.

IV. Data and Analysis

In this section, we present the results of our study and the data analyses considering our research question. (We discuss further implications for the data and results in Section V.) The research question considers whether our approach can be efficient in reducing efforts to generate new test cases during regression testing. To answer this question, we compare the results collected with and without using reusable input values. To do so, we gather three sets of data: (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.

Table IV summarizes the data we collected. In addition to these three data sets, we also provide an overview about how many test paths are required to test modified versions of the applications[1].

The table lists, for each application, the "Version Pair" (two versions of the application analyzed), "Total Number of Path" (the total number of test paths required for the new version), "Regression Paths" (the number of test paths required for the areas affected by code changes in the new version), "Total Input Values" (the total number of input values required for the new test paths), "Reusable Input Values" (the number of reusable input values), and "Input Reuse Rates" (the percentage of input values that can be reused over the control technique). We show the version pair data because regression testing starts with the second release of the application. Figure 3 visually shows the input value reuse rates for all version pairs for five applications.

When we do not consider code changes, for the latest version of the five applications (in order of their appearance in the table), 73, 2409, 529, 1444, and 4419 test paths are required. While we can reduce a large number of test paths when we only consider testing the areas affected by changes, we can see that we still require a large number of test paths. (e.g., for *osCommerce* 2.2MS2, 1719 paths are needed; for *Mantis* 1.2.0, 2802 paths are needed.)

Now, we discuss the results for each application.

[1]There were a few bugs in the implementation of path generation algorithm in our previous work which were fixed before collecting data in this study. Due to this reason, the data values presented in this paper do not match with those published in our previous work [1].

A. Results for FAQForge

Table IV shows that, for the version pair 1.3.0 and 1.3.1, the total number of required input values is 25. Among these 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 applications, and the changes between versions are also very small; therefore the number of input values required for the new paths is relatively small, and this result is not surprising.

By inspecting the files in *FAQForge*, we found that only one of the files in version 1.3.1 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.

The second version pair of *FAQForge* (1.3.1 and 1.3.2), unlike the first version 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 number of input values was 210, and among them, 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.

To understand the low number for reusability, we examined the second version pair. We found that most changes in the source files were 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, the number of reusable variables was very low.

As shown in Table IV and Figure 3, our technique required a relatively small number of test input values compared to the control technique. Our technique was able to reuse 76% and 7% of the input values for versions 1.3.1 and 1.3.2, respectively.

B. 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 (2.2MS1 and 2.2MS2), we 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.

From the table, we can see that the first version pair needs to solve 4392 input values to execute all the regression test paths. Among these values, 702 are reusable for regression testing the new version of source code.

For the second version pair, 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 had to be solved to execute all the regression test paths. Among these values, 219 were reusable for regression testing 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. Our technique was able to reuse input values 16% and 27% for versions 2.2MS2 and 2.2MS2-060817, respectively.

C. Results for phpScheduleIt

The *phpScheduleIt* results showed slightly lower reusabilities compared to other applications. The first two version pairs

TABLE IV: Total Paths, Regression Paths, Total Inputs, Reusable Inputs, and Input Reuse Rates

Application	Version Pair	Total Number of Paths	Regression Paths	Total Input Values	Reusable Input Values	Input Reuse Rates
FAQForge	1.3.0 & 1.3.1	73	5	25	19	76%
	1.3.1 & 1.3.2	73	19	210	15	7%
osCommerce	2.2MS1 & 2.2MS2	2403	1719	4392	702	16%
	2.2MS2 & 2.2MS2-060817	2409	58	813	219	27%
phpScheduleIt	1.0.0 & 1.1.0	481	338	2389	280	12%
	1.1.0 & 1.2.0	518	314	3877	472	12%
	1.2.0 & 1.2.12	529	154	2339	861	37%
Mambo	4.5.5 & 4.5.6	1357	65	490	294	60%
	4.5.6 & 4.6.1	1388	1388	4446	67	2%
	4.6.1 & 4.6.2	1416	236	2075	536	26%
	4.6.2 & 4.6.3	1409	92	1236	250	20%
	4.6.3 & 4.6.4	1444	114	1567	191	12%
	4.6.4 & 4.6.5	1444	20	324	57	18%
Mantis	1.1.6 & 1.1.7	3482	221	1524	708	46%
	1.1.7 & 1.1.8	3482	185	1238	662	53%
	1.1.8 & 1.2.0	4345	2802	20425	195	1%
	1.2.0 & 1.2.1	4373	166	1772	558	31%
	1.2.1 & 1.2.2	4389	106	1042	296	28%
	1.2.2 & 1.2.3	4403	103	890	267	30%
	1.2.3 & 1.2.4	4419	199	2643	263	10%

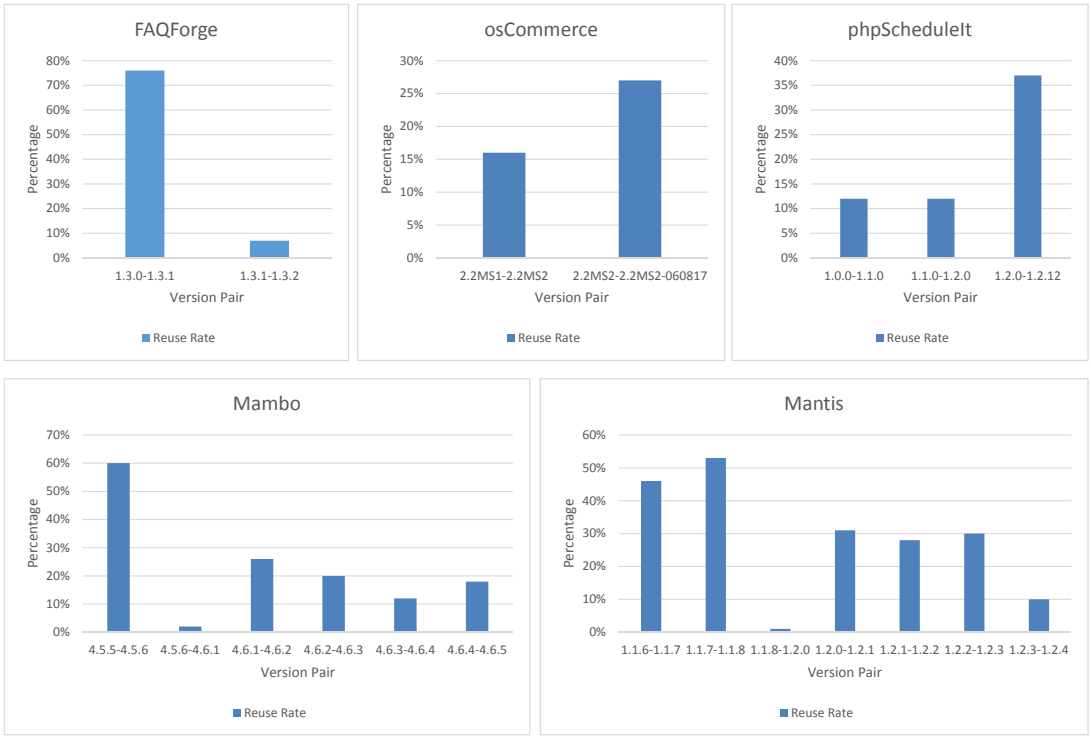

Fig. 3: Input Value Reuse Rates

showed an average 12% reuse rate while the last version pair showed a 37% reuse rate.

Upon further investigation, we found that the first two version pairs involved large functionality changes and bug fixes. The majority of the generated regression test paths for those version pairs were incompatible. As a result, only a small number of variables were eligible for reuse. In the case of the final version pair (1.2.0 and 1.2.12), only a few source code files changed. The release note for version 1.2.12 indicated that an update from version 1.2.0 to 1.2.12 involved minor patches and a few bug fixes.

D. Results for Mambo

As we can easily observe from Figure 3, the results for *Mambo* varied across six version pairs. In particular, for version pair 4.5.6 and 4.6.1, only 2% of the variables were reusable. Upon manual inspection of these two versions, there were a large number of file changes. The release note for version 4.6.1 also reported that it had a major update with lots of feature changes and bug fixes. Again, we think that a large amount of code changes for this version pair resulted in a low input value reuse rate.

Other version pairs also contained many changes. In fact, none of the versions are not small patch updates or small bug fixes. The first version pair (4.5.5 and 4.5.6) had relatively small code changes compared to other version pairs, so it produced the highest input value reuse rate (60%). The rest of the version pairs produced on average 19% reuse rate.

E. Results for Mantis

Similar to the *Mambo* results, the results for *Mantis* varied across seven version pairs.

The third version pair, 1.1.8 and 1.2.0, showed a very low reuse rate. Only 1% of the input values were reusable for regression test paths of version 1.2.0. Manual source code inspection for these two versions revealed that a large number of files changed from version 1.1.8 to version 1.2.0. Further, the release note for version 1.2.0 reported that it went through a lot of feature updates and bug fixes. These major updates resulted in generating a large number of regression test paths as well as a low number of reusable input values.

Other version pairs showed, on average, 33% reusability. The first two version pairs (1.1.6 and 1.1.7, and 1.1.7 and 1.1.8) produced nearly 50% of the input value reuse rate. Again, as we observed from other applications' results, version pairs with small patch updates or bug fixes could gain substantial benefits by using our input value reuse approach.

V. Discussion

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 1719 for version 2.2MS2. For those test paths, we learned from our 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 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 *phpScheduleIt*, 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 *Mombo* 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 very 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 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.

VI. 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., [13], [14], [15], [16]). Existing test cases, however, 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 [17], [18], [19], [20], [21], [22]. Apiwattanapong et al. [17] and Santelices and Harrold [18] 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. [19], [20] 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. [21] proposed an efficient test generation technique that uses dynamic symbolic execution, eXpress, by pruning irrelevant paths. Chen et al. [22] proposed a model-based regression test suite generation using dependence analysis. Rubinov and Wuttke [23] presented a framework for augmenting test suites automatically by using information obtained from unit test cases. These approaches focused on desktop applications such as C or Java, but our approach applied regression testing to web applications, creating different challenges (e.g., dynamic programming languages deployed using multi-tier architecture). Further, our major focus in this paper was 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. [24] and Artzi et al. [25] 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 [26] used a crawler to discover the link structure of web applications and to produce test case specifications from this structure. Deng et al. [27] utilized static analysis to extract information related to URLs and their parameters for web applications, as well as to generate executable test cases using the collected information. Halfond et al. [28] presented an approach that uses symbolic execution to identify precise interfaces for web applications.

While our work 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 [29], [30] has been done, but their focus was different than ours. Dobolyi and Weimer [29] presented an approach that automatically compares the outputs from similar web applications to reduce the regression testing effort. Elbaum et al. [30] presented a web application testing technique that utilizes users session data considering regression testing contexts. Alshahwan and Harman [31] proposed a new test generation approach that uses output uniqueness to make test suite augmentation more effective.

Another research area that is slightly relevant to our work is constraint resolution. Many researchers have worked on this area [2], [32], [33], [34], and now, several constraint resolution tools are available, including Hampi [6] and Choco [7], which we used in our previous work. Although these automatic resolution tools helped us resolve 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 effort required to resolve them, which motivated this research (collecting reusable constraint input values). We envision that, by combining these two approaches (automatic resolution using constraint solvers and our approach that was proposed in this paper), we can expect great effort savings during regression testing.

VII. Conclusions and Future Work

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

Further, the constraints and actual values for variables may be reusable across several versions as long as the definition and use relationships of the variables hold across versions. This means that we may expect greater savings as the applications evolve over time.

While our approach may reduce the effort needed to apply regression testing for patched web application software, it may also reduce testing 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, as we mentioned in Section VI, by combining our approach with automatic resolution using constraint solvers

(for instance, feeding initial input values to the solvers by analyzing existing executable test cases), we may further accelerate regression testing processes.

The results of our studies suggest some future work. First, we evaluated our approach using five widely used open source web applications with three versions. While the experiment results are promising, we investigated our approach using a relatively small number of versions for both applications. This means that the experiment results we obtained in this study do not sufficiently capture possible benefits and factors related to the long-term utilization of our technique. Certainly, we believe that our approach would provide greater benefits when we apply it 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).

We also plan to perform additional studies that apply our approach to a wider population (e.g., larger open source and industrial-size applications) with different testing processes (e.g., constraints imposed by an industry's regression testing practice). Currently, we are preparing larger open source web applications with several versions written in PHP.

Further, we utilized existing test cases to extract reusable variables, their constraints, and their input values, but we did not use the actual existing test cases 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, we may achieve additional savings. Thus, we plan to investigate test case selection approaches that choose test cases that exercise the modified areas code to help reduce the cost of generating new tests.

Acknowledgments

Mike Delaney, Justin Anderson, Joshua Tan, Cesar Ramirez, and Aaron Marback helped construct parts of the tool infrastructure used in the experimentation. This work was supported, in part, by NSF Awards CNS-0855106 and CCF-1050343, and NSF CAREER Award CCF-1149389 to North Dakota State University.

REFERENCES

[1] A. Marback, H. Do, and N. Ehresmann, "An effective regression testing approach for PHP web applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, Apr. 2012, pp. 221–230.

[2] P. Hooimeijer and W. Weimer, "Solving string constraints lazily," in *Proceedings of the 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," in *Proceedings of the International Conference on Software Engineering*, May 2009, pp. 199–209.

[4] W. Visser, J. Geldenhuys, and M. 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*, ser. FSE '12, 2012, pp. 58:1–58:11.

[5] phc, http://www.phpcompiler.org/.

[6] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, and M. Ernst, "HAMPI: a solver for string constraints," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2009, pp. 105–115.

[7] Choco, http://www.emn.fr/z-info/choco-solver/.

[8] osCommerce, http://www.oscommerce.com/.

[9] FaqForge, http://sourceforge.net/projects/faqforge/.

[10] phpScheduleIt, http://www.php.brickhost.com/.

[11] Mambo, http://www.mamboserver.com/.

[12] Mantis, http://www.mantisbt.org/.

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

[14] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, Apr. 1997.

[15] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2006, pp. 1–12.

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

[17] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold, "MATRIX: Maintenance-oriented testing requirements identifier and examiner," in *TAIC PART*, 2006.

[18] R. Santelices and M. J. Harrold, "Applying agressive propagation-based strategies for testing changes," in *International Conference on Software Testing, Verification and Validation*, Apr. 2011.

[19] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2010.

[20] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 2011.

[21] K. Taneja, T. Xie, N. Tillmann, and J. Halleux, "eXpress: Guided path exploration for efficient regression test generation," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2011.

[22] Y. Chen, R. Probert, and H. Ural, "Model-based regression test suite generation using dependence analysis," in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, Jul. 2007.

[23] K. Rubinov and J. Wuttke, "Augmenting test suites automatically," in *International Conference on Software Engineering, Posters and Information Tool Demonstrations*, Jun. 2012, pp. 1433–1434.

[24] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2008.

[25] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proceedings of the International Conference on Software Engineering*, May 2010.

[26] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 25–34.

[27] Y. Deng, P. Frankl, and J. Wang, "Testing web database applications," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–10, 2004.

[28] W. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2009.

[29] K. Dobolyi and W. Weimer, "Harnessing web-based application similarities to aid in regression testing," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 2009.

[30] S. Elbaum, S. Karre, and G. Rothermel, "Improving web application testing with user session data," in *Proceedings of the Fourteenth International Conference on Software Engineering*, 2003, pp. 49–59.

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

[32] N. Klarlund, "Mona Fido: The logic-automaton connection in practice," in *Computer Science Logic*, 1998, pp. 311–326.

[33] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2007.

[34] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of PLDI*, Jun. 2007.