Adaptive Regression Testing Strategy: An Empirical Study

Md. Junaid Arafeen* and Hyunsook Do*
*Department of Computer Science
North Dakota State University
Fargo, ND
{md.arafeen, hyunsook.do}@ndsu.edu

***Abstract*—When software systems evolve, different amounts and types of code modifications can be involved in different versions. These factors can affect the costs and benefits of regression testing techniques in different ways, and thus, there may be no single regression testing technique that is the most cost-effective technique to use on every version. To date, many regression testing techniques have been proposed, but no research has been done on the problem of helping practitioners systematically choose appropriate techniques on new versions as systems evolve. To address this problem, we propose adaptive regression testing (ART) strategies that attempt to identify the regression testing techniques that will be the most cost-effective for each regression testing session considering organization's situations and testing environment. To assess our approach, we conducted an experiment focusing on test case prioritization techniques. Our results show that prioritization techniques selected by our approach can be more cost-effective than those used by the control approaches.**

***Keywords*-Regression testing, adaptive regression testing strategy, test case prioritization, Analytical Hierarchy Process**

I. Introduction

Regression testing is an important and necessary activity that can maintain the quality of modified software systems. To date, many regression testing techniques have been proposed. For instance, regression test selection techniques (e.g., [1], [2], [3]) reduce testing costs by selecting test cases that are necessary to test a modified program. Test case prioritization techniques (e.g., [4], [5], [6]) reorder test cases, scheduling test cases with the highest priority according to some criterion earlier in the testing process to yield benefits such as providing earlier feedback to testers and earlier fault detection.

While this research has made considerable progress in regression testing areas, one important problem has been overlooked. As systems evolve, the types of maintenance activities that are applied to them change. Differences between versions can involve different amounts and types of code modifications, and these changes can affect the costs and benefits of regression testing techniques in different ways. Thus, there may be no single regression testing technique that is the most cost-effective technique to use on every version. For instance, as we observed from our study, prioritization technique that works best changes across versions.

We propose to address this lack by creating and empirically studying *adaptive regression testing* (ART) strategies. ART strategies are approaches that operate across system lifetimes, and attempt to identify the regression testing techniques that will be the most cost-effective for each regression testing session. ART strategies evaluate regression testing techniques in terms of decision criteria such as cost and benefit factors and choose the best alternative among techniques considering organization's situations and feedback from prior regression testing sessions.

The problem of performing such evaluations is known as the "multiple criteria decision making" (MCDM) problem, and MCDM approaches have been used in many science, engineering, and business areas that involve complex decision problems, such as technology investment, resource allocation, and layout design [7], [8]. To date, many MCDM approaches have been proposed including the Weighted Sum Model (WSM), the Weighted Product Model (WPM), the Analytical Hierarchy Process (AHP), and other variants. Among these MCDM methods, AHP has been one of the more popular methods, having been used by researchers and practitioners in various areas including software engineering [9], [10], [11].

Therefore, in this research, as an initial approach to creating ART strategies, we investigated AHP method [7] to see whether AHP can be effective for selecting appropriate regression testing techniques across system lifetime, particularly focusing on test case prioritization techniques. To do this, we have designed and conducted a controlled experiment using several Java programs with multiple versions considering several selection strategies. The results of our experiment show that the prioritization techniques selected by AHP can be more cost-effective than those used by the control approaches.

In the next section, we describe background information and related work relevant to prioritization techniques and regression testing strategies. Section III describes our proposed approach, ART strategy. Section IV presents our experiment setup, Section V presents results and analysis, and Section VI addresses threats to validity. Section VII discusses our results, and Section VIII presents conclusions and future work.

II. Background and Related Work

Regression testing attempts to validate modified programs to see whether changes have produced unintended effects. Depending on various factors, such as the size and complexity of the program and its test suite, regression testing process can be very expensive. Thus, many researchers have proposed numerous cost-effective regression testing techniques including regression test selection, test suite reduction/minimization, and test case prioritization, but here, we limit to our discussion to test case prioritization, which is directly related to our work.

Test case prioritization techniques (e.g., [12], [6]) reorders test cases in order to increase the chance of early fault detection using various types of information available from software artifacts, such as the coverage of code achieved by tests, code change information, or code complexity. For example, one technique, *total block coverage prioritization*, simply sorts the test cases in the order of the number of blocks they cover. One variation of this technique, *additional block coverage prioritization* iteratively selects a test case that yields the greatest block coverage, then adjusts the coverage information for the remaining test cases to indicate their coverage of blocks not yet covered, and then repeats this process until all blocks coverable by at least one test case have been covered.

To date, numerous test case prioritization techniques have been proposed, and a recent paper by Yoo and Harman [13] provides a comprehensive overview of these techniques. While the goal of the proposed techniques is to improve the effectiveness of regression testing, to be useful in practice, techniques should be applicable within various testing environments and contexts. Recent research on test case prioritization has employed empirical studies to evaluate the cost-benefit tradeoffs among techniques considering various factors and testing contexts [14], [15], [16], [17], [18]. For instance, Do et al. [14] and Walcott et al. [18] have studied the effects of time constraints imposed on regression testing through empirical studies. Qu et al. [17] consider prioritization in the context of configurable systems.

Studies such as these have allowed researchers and practitioners to understand factors that affect the assessment of techniques and to compare techniques in terms of costs and benefits relative to actual software systems. However, studies to date have not considered strategies for selecting appropriate techniques under particular circumstances as systems evolve. Only few studies [19], [20] have done on the problem of helping practitioners choose appropriate techniques under particular system and process constraints. Harrold et al. [20] present empirical results that demonstrate how code modifications can affect the choice of regression test selection methods. Elbaum et al. [19] perform experiments exploring characteristics of program structure, test suite composition, and changes on prioritization, and identified several metrics characterizing these attributes that correlate with prioritization effectiveness. The empirical results of their study provide insights into which prioritization technique is appropriate (or not appropriate) under specific testing scenarios. Unlike our approach, these two studies evaluate techniques solely relied on software metrics and did not consider the notion of software evolution context.

Since many factors can be involved in evolving systems, selecting appropriate techniques for each version can be a multiple criteria decision making (MCDM) problem. Analytic Hierarchy Process (AHP) is one of the widely used MCDM methods, and many areas that involve complex decision problems, such as business, manufacturing, science and engineering. For instance, Kamal and Al-Harbi [21] use AHP in project management to determine the contractors' competence or ability to participate in the project bid. AHP has also been used in determining the best manufacturing system [22], layout design [23], and the evaluation of technology investment decisions [24].

Recently AHP has been used in software engineering areas. Barcusa and Montibellerb [25] use AHP to allocate software development work in distributed teams. They develop a multi-criteria decision model to support the distributed team work allocation decision by using decision conferencing and multi-attribute value analysis. Finnie et al. [26] use AHP to prioritize software development productivity factors, and Karlsson et al. [9] and Perini et al. [27] compare AHP with other alternative method in prioritizing software requirements. Yoo et al. [11] use AHP to improve test case prioritization techniques by employing expert knowledge, and compare the proposed approach with the conventional coverage-based test case prioritization technique. Unlike their study, in this paper, we utilize AHP to develop adaptive regressions testing strategy, which helps identify the best test case prioritization techniques across system lifetime.

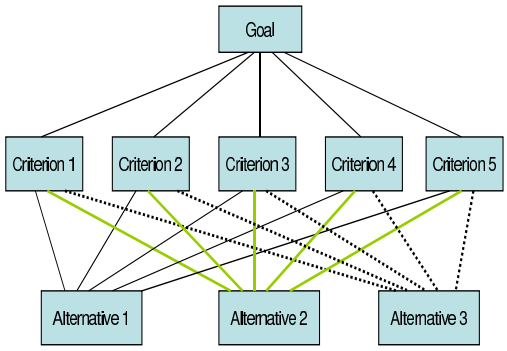
III. Adaptive Regression Testing (ART) Strategy

In this section, we describe AHP method and how AHP is used for creating ART strategy using an example.

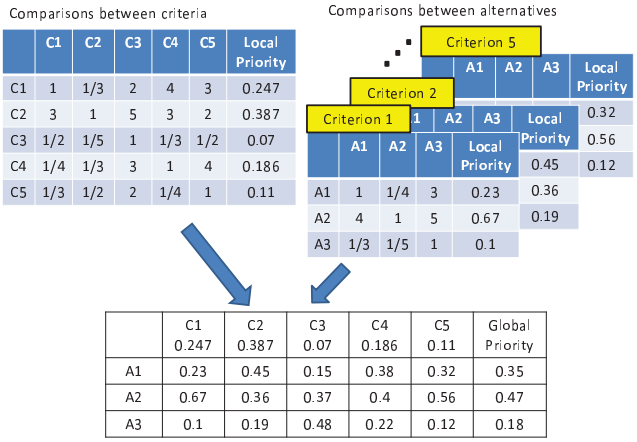
A. AHP Method

To use AHP, decision makers first define a hierarchy that describes the problem they want to solve. As shown in Figure 1.a, adapted from [7], an AHP hierarchy consists of a goal that they want to achieve, alternatives that are available to reach the goal, and criteria that are factors that may be used in decision making about these alternatives. The criteria can be further partitioned into sub-criteria if necessary.

Once decision makers define an AHP hierarchy, two types of pairwise comparisons are performed: between pairs of criteria and between pairs of alternatives as shown in Figure 1.b. When comparing pairs of criteria (the upper left table), decision makers assign relative importance weights to criteria; for example, C1 is given importance 4 relative



(a) An AHP hierarchy



(b) An AHP Example

Figure 1. An AHP hierarchy and an example application of AHP.

to C4. After completing this matrix, the assigned values are normalized and the local priority of each criterion is produced, which is shown in the rightmost column of the table (and in the top row of the bottom table in the figure). The local priority is calculated by the following equation: $LP_i = \frac{\sum_{j=1}^{N}(RW_{ij})}{\sum_{i=1}^{N}\sum_{j=1}^{N}(RW_{ij})}$, where LP_i is a local priority of criterion i, RW_{ij} is a relative weight of criterion i over criterion j, and N is the number of criteria (The local priorities of alternatives are calculated in the same way).

Similarly, matrices that show the relative importance of alternatives for each criterion are constructed. In this example, five matrices are constructed because there are five criteria (the upper right tables). Again, the assigned values are normalized for each matrix, and local priorities are produced for each alternative (the resulting local priorities appear in the bottom table in the figure).

After calculating the local priorities for criteria and alternatives, an M x N matrix is constructed, as shown in the bottom table in Figure 1.b, where M is the number of alternatives considered and N is the number of criteria. In our example, M is 3 and N is 5. Then, weighted sums of the values per technique are calculated; these are shown in the rightmost column ("global priorities"i). The global priority is calculated by the following equation: $GP_k = \sum_{j=1}^{N}(LPA_{kj}) * (LP_j))$, where GP_k is a global priority for alternative k, N is the number of criteria, LPA_{kj} is a local priority of alternative k ($1 \leq k \leq M$, M is the number of alternatives) for criterion j, LP_j is a local priority of criterion j. Based on the weighted sum values, decision makers can determine which alternative should be selected. In this example, *T2* (0.47) performs best and *T1* (0.35) next best, with *T3* (0.18) far behind.

B. Applying AHP to Prioritization Strategy

We now describe how AHP is applied to prioritization strategy that we use in this work. While we describe this in terms of test case prioritization using one of the programs we used in our study, the approach could be applied to any regression testing techniques and any system for which the required information is available.

As outlined in the prior section, to apply AHP to prioritization strategy, the following steps are required:

1. Set a goal
2. Identify alternatives that are available to reach the goal
3. Identify evaluation criteria for alternatives
4. Pairwise comparisons: between pairs of criteria and between pairs of alternatives
5. Obtain global priorities of alternatives

The following subsections describe each of these in detail.

1) Step 1: Set a Goal: Suppose that the goal of test engineers is to choose the most cost-effective test case prioritization technique in application to a particular system version.

2) Step 2: Identify Alternatives: To achieve this goal, test engineers consider several different types of prioritization techniques as alternatives. For instance, test engineers could consider traditional coverage-based test case prioritization techniques, such as total block coverage based test case prioritization, and additional block coverage based test case prioritization.

3) Step 3:Identify Evaluation Criteria: As criteria, test engineers choose factors that are influential in evaluating test case prioritization techniques. For instance, test engineers could consider the cost factors that can affect the choice of techniques, such as the cost of applying test case prioritization technique or the cost of software artifact analysis.

4) Step 4: Pairwise Comparisons: Next, two types of pairwise comparisons are performed: between pairs of criteria and between pairs of techniques as we explained in Section III-A. To do so, test engineers assign relative importance weights to criteria and techniques using the scale of weights they define. In this step, test engineers rely on

their experiences and history data regarding the performance of test case prioritization techniques.

5) Step 5: Obtain Global Priorities: Once test engineers assign relative weights, global priorities are calculated as explained in Section III-A and this step can be automated by building an AHP tool. Based on global priorities, test engineers determine which technique they should use for the particular version of the program.

Steps 2 and 3 are dependent on an organization's testing practices and environment. Figure 2 summarizes steps 4 and 5 graphically. As we can see from the figure, the test engineer examines history data and various software artifacts for the current version, and assigns relative weights for criteria and techniques. This process requires human judgment, so it is done manually. In practice, often organizations rely on human experts' opinions or experienced members' judgment when they make important technical decisions (e.g., which tools or techniques should be used), so this is not an uncommon process in software industry.

The rest of the processes can be automated. The AHP tool takes relative weights of criteria and techniques, and produces matrices shown in Figure 1. Then, finally the test engineer can decide which technique should be used based on global priorities that the tool produced.

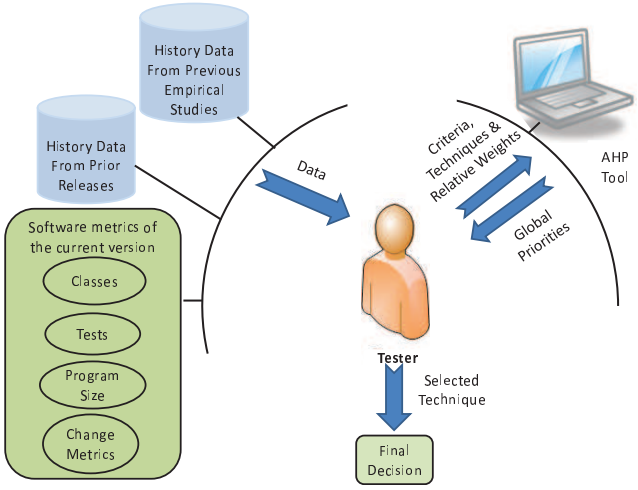


Figure 2. AHP Process

IV. Empirical Study

In this study, we address the following research question:

RQ: Is AHP effective for selecting appropriate test case prioritization techniques across system lifetime?

To investigate the research question, we performed a controlled experiment. The following subsections present, for this experiment, our objects of analysis, variables and measures, experiment setup and design, and threats to validity. Following this presentation, in Section V we present our data and analysis, in Section VI we address threats to validity, and in Section VII we discuss practical implications of the results.

A. Objects of Analysis

We considered five Java programs obtained from the SIR infrastructure [28] as our objects of analysis: *ant*, *xml-security*, *jmeter*, *nanoxml*, and *galileo*. *Ant* is a Java-based build tool, *jmeter* is a load testing tool for client/server application, and *xml-security* provides security functionality for XML data. *Nanoxml* is a small XML parser for Java, and *galileo* is a Java bytecode analyzer. Several sequential versions of each of these programs are available. The first three programs are provided with JUnit test suites, and the last two are provided with TSL (Test Specification Language) test suites [29].

Table I lists, for each of our objects of analysis, data on its associated "Versions" (the number of versions of the object program), "Classes" (the number of class files in the latest version of that program), "Size (KLOCs)" (the number of lines of code in the latest version of the program), and "Test Cases" (the number of test cases available for the latest version of the program). To study the research question we require fault data, so we utilized mutation faults provided with the programs [30]. The rightmost column, "Mutation Faults", is the total number of mutation faults of the program (summed across all versions).

Table I
Experiment Objects and Associated Data

Objects	Versions	Classes	Size (KLOCs)	Test Cases	Mutation Faults
ant	9	914	61.7	877	412
jmeter	6	434	42.2	78	386
xml-sec.	4	145	15.9	83	246
nanoxml	6	64	3.1	216	204
galileo	16	68	14.5	912	2494

B. Variables and Measures

1) Independent Variable: To investigate our research question we manipulate one independent variable: *test case prioritization technique application mapping strategy*, which assigns, to a specific sequence of versions $S_i, S_{i+1}, \ldots S_j$ of system *S*, specific test case prioritization techniques. As test case prioritization techniques we utilize original order (Orig: the order in which test cases are executed in the original testing scripts provided with the object programs), random order (Rand: in our experiment, averages of runs of 30 random orders), and two test case prioritization heuristics (total block coverage (Tcov) and additional block coverage (Acov) prioritization techniques explained in Section II).

We consider five mapping strategies as follows:

- Tcov-all: Use of the total coverage technique across versions (a control)

- Acov-all: Use of the additional coverage technique across versions (a second control)
- Rand-all: Use of the random technique across versions (a third control)
- Orig-all: Use of the original technique across versions (a fourth control; it is used as a baseline strategy)
- AHP: Evolutionary adaptation of techniques following the AHP method described in Section III. The AHP method selects the best technique among four prioritization techniques (Tcov, Acov, Rand, and Orig) for each version based on the criteria we identifies and expert's opinion. More details on how we applied AHP are described in Section IV-C.

2) Dependent Variable and Measures: Our dependent variable is a *relative cost-benefit value* produced by applying EVOMO economic model presented in [31], using a further calculation described below (Equation 1). The cost and benefit components are measured in dollars. To determine the *relative cost-benefit* of prioritization technique *T* with respect to baseline technique *base*, we use the following equation:

$$(\text{Benefit}_T - \text{Cost}_T) - (\text{Benefit}_{base} - \text{Cost}_{base}) \quad (1)$$

When this equation is applied, positive values indicate that T is beneficial compared to $base$, and negative values indicate otherwise. We used the original technique as a baseline in this experiment. This means that the Orig-all strategy functions as a baseline strategy when we consider the cost-benefit values across all versions of the program.

EVOMO[1] involves two equations as shown in Equations 2 and 3: one that captures costs related to the salaries of the engineers who perform regression testing (to translate time spent into monetary values), and one that captures revenue gains or losses related to changes in system release time (to translate time-to-release into monetary values).

$$Cost = salary * \sum_{i=2}^{n} (setup(i) + obsoleteTests(i) + resultValidation(i) + missedFaults(i)) \quad (2)$$

$$Benefit = revenue * \sum_{i=2}^{n} (deliveryTime(i) - (setup(i) + obsoleteTests(i) + analysis(i-1) + runTechinique(i) + testExecution(i) + resultsValidation(i) + faultDetectionDelay(i))) \quad (3)$$

Significantly, the model accounts for costs and benefits across entire system lifetimes, rather than on snapshots (i.e. single releases) of those systems, through equations that calculate costs and benefits *across entire sequences of system releases*. The major cost components that EVOMO captures are as follows: costs for applying regression testing techniques, costs associated with missed faults, costs for artifact analysis, costs of delayed fault detection feedback, and costs associated with obsolete tests.

[1]Here we just summarize each equation briefly. See [31], [32] for detailed descriptions.

C. Experiment Setup and Procedure

To measure costs of delayed fault detection feedback and costs for applying regression testing techniques, we required object programs containing faults. Similar to our early studies [14], [32], to obtain the fault data required to investigate our research question, for each version of each program we randomly selected a *mutant group* from the set of that version's mutation faults. Each mutant group contained at most 10 mutants.

To apply AHP, we followed steps described in Section III. As a human tester, one graduate student who has three years of software industry experience performed the AHP processes. The student considered the following criteria to evaluate prioritization techniques:

- Cost of applying test case prioritization technique: the time required to run a test case prioritization algorithm
- Cost of software artifact analysis: the costs of instrumenting programs and collecting test execution traces
- Cost of delayed fault detection: the waiting time for each fault to be exposed while executing test cases under a test case prioritization technique
- Cost of missed fault: the time required to correct missed faults

Next, the student performed pairwise comparisons using the scale of weights as shown in Table II, which has been commonly used by others [7], [11]. When the student assigned relative weights, he utilized history data regarding the performance of test case prioritization techniques observed from previous several empirical studies [14], [15], [32], [33].

To support the rest of the processes, we implemented a Java Swing-based AHP tool. The AHP tool takes relative weights of criteria and techniques, and produces local and global priorities based on the AHP algorithm [7]. Finally the student determined which technique should be used for each version of the program using global priorities.

Table II
SCALE OF WEIGHTS

Weight	Definition of Weight
1	equally important
3	moderately important
5	strongly important
7	very strongly important
9	extremely important

Often software companies have time pressure with the product release, due to the constraint budgetary problem and competitive software market. In practice, situations in which time constraints intervene to affect product release are frequent in the software industry, and typically software

companies cut back on testing activities in order to ensure timely release of their product. Further, the degree of time constraints can vary as systems evolve. For instance, for a certain release, a company could suffer more time constraints compared to other releases due to the complex feature addition or the technical personnel loss. Thus, in this experiment, we consider the situation with time constraints that vary with each version when we evaluate test case prioritization techniques.

To simulate this situation, for each of the test case prioritization techniques, we randomly assigned the level of time constraints (25%, 50%, or 75%) for each version, and foreshortened the test execution process for each version by the assigned time constraint level. We ran four sets of random assignments across all versions for each program, applied the AHP processes we just explained, and collected cost-benefit values for all strategies.

V. Data and Analysis

In this section, we present the results of our study. We summarize the data in Table III. Table III shows experiment results that were collected by running four sets of random assignments (run 1 through run 4 in the table) of three time constraint levels for each version of the program. Since the Orig-all strategy is the baseline used in our relative cost-benefit calculation, results for that strategy are not shown explicitly in the tables.

Table III contains five subtables, and each subtable corresponds to the results of each program. All of the data in these tables shows the relative cost-benefit value in dollar with respect to the baseline technique (Orig) as defined in Section IV-B2. Higher values indicate greater cost-benefits. Within each subtable in the tables, columns are labeled with four runs, for each run listing four test case prioritization testing strategies. Rows are labeled with versions of the program and the last row ("Total") shows the sum of the cost-benefit values for all versions . Now, we describe each of these subtables.

The first subtable shows the results for *ant*. The results vary across versions, but the total cost-benefit values indicate that the prioritization techniques selected by AHP were more cost-effective than those used by the control strategies except for one case (Rand-all in run 3 was better than AHP). In particular, the cost-benefit value gap between the AHP strategy and the two control strategies (Tcov-all and Acov-all) is large, and Tcov-all was even worse than the baseline strategy in some cases (run 1 and run 4). Among the control strategies, Rand-all produced the best results.

The second subtable shows the results for *jmeter*. Similar to the results on *ant*, the AHP strategy was more cost-effective than the control strategies except for run 2. Among the control strategies, Acov-all produced the best results, Rand-all performed relatively well, but Tcov-all was even worse than the baseline strategy in most cases.

The third subtable shows the results for *xml-security*. As the results show, the AHP strategy was more cost-effective than the two control strategies (Tcov-all and Rand-all), but it was not better than the Acov-all strategy. Unlike the results on *ant* and *jmeter*, Tcov-all produced better results than the baseline strategy.

The fourth subtable shows the results for *nanoxml*. Overall, the AHP strategy outperformed all control strategies except for one case (Acov-all in run 2 was better than AHP). Similar to the results on *xml-security*, Tcov-all produced better results than the baseline strategy.

The last subtable shows the result for *galileo*. The results show that the AHP strategy was more cost-effective than control strategies except for one case (Acov-all in run 3 was better than AHP). Tcov-all produced worse results than the baseline strategy in all cases.

The total cost savings across all versions are one measure that shows the effectiveness of the strategies, but this measure can be misleading because abnormal cost-benefit values for particular version could affect the entire outcome. Thus, we examined how often the strategies produce the best results across all versions.

Figure 3 presents bar graphs of the results. The figure contains five subfigures that present results for each of the object programs, and each subfigure contains bar graphs for four prioritization strategies showing the total number versions that produced the best results by those strategies, for the given object program and four runs. For instance, in run 1 for *ant*, Tcov-all performed best for one version (version 1 in Table III) and Acov-all performed best for two versions (versions 2 and 4 in Table III).

Overall, the AHP strategy produced the best results (16 out of 20 cases were better than the control strategies – in total, we have the 20 observed data points.) and the Acov-all strategy performed relatively well compared to other control strategies (9 out of 20 cases performed best), but the trend varied across programs.

In the cases of *ant* and *nanoxml*, the AHP strategy was consistently better than all three strategies across all runs with one exception. In the case of *xml-security*, Acov-all was slightly better than AHP.

Comparing the control strategies, Rand-all outperformed others in *ant*, and it was even better than AHP for one case (run 3). However, in other cases, Rand-all did not perform well. In particular, in the case of *xml-security*, Rand-all did not produce any single best result. Similar to the results we observed in Table III, Tcov-all performed worst. Only in three programs (*ant*, *jmeter*, and *xml-security*), it produced the best result for one version for six out of 12 cases. In other programs, it did not produce any single best result.

Overall, the trends we observed from this figure are consistent with those we observed from Table III, but we also found some differences. While AHP outperformed 15 out 20 cases when we considered the total cost-benefit values,

Table III
Experiment Results: Relative Cost-Benefit Values (dollars)

ant																
	Run 1				*Run 2*				*Run 3*				*Run 4*			
	Tcov -all	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*
v1	135	77	-40	-40	367	232	102	102	-19	-70	-98	-98	135	77	-40	-40
v2	205	209	139	209	205	209	139	209	207	209	79	209	55	326	161	326
v3	-58	-62	48	48	-151	92	49	49	-58	-62	48	48	-59	91	46	46
v4	-66	14	0	14	-155	-72	-58	-58	12	13	42	13	-66	14	0	14
v5	-99	-133	26	26	-157	-191	18	18	-99	-133	26	26	-145	-179	32	32
v6	-142	-180	7	7	-142	-180	7	7	-37	-113	87	-37	337	407	560	407
v7	-160	-201	32	32	275	234	324	324	-142	-183	48	48	-160	-201	32	32
v8	-107	-248	146	146	-107	-248	146	146	143	116	292	292	-128	115	215	215
Total	-292	-524	358	442	135	76	727	797	7	-223	524	501	-31	650	1006	1032

jmeter																
	Run 1				*Run 2*				*Run 3*				*Run 4*			
	Tcov -all	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*
v1	15	17	50	17	47	135	180	135	15	17	50	50	-73	116	97	116
v2	-51	153	93	153	-85	-85	-6	-85	-51	153	93	153	-51	153	93	153
v3	130	266	277	277	130	266	277	277	-66	22	-36	22	-66	22	-36	22
v4	121	31	5	121	-64	-65	-142	-64	35	274	5	274	35	274	5	274
v5	-196	-196	-135	-196	-174	-144	-136	-174	-174	-144	-136	-174	-196	-196	-135	-196
Total	19	271	290	372	-146	107	173	89	-241	322	-24	325	-351	369	24	369

xml-security																
	Run 1				*Run 2*				*Run 3*				*Run 4*			
	Tcov -all	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*
v1	177	274	88	274	37	38	6	38	268	331	203	331	37	38	6	38
v2	26	117	-44	117	26	117	-44	117	-48	14	-190	14	26	117	-44	117
v3	170	170	71	170	499	546	315	546	170	170	71	170	499	546	315	499
Total	373	561	115	561	562	701	277	701	390	515	84	515	562	701	277	654

nanoxml																
	Run 1				*Run 2*				*Run 3*				*Run 4*			
	Tcov -all	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*
v1	931	966	975	975	928	962	860	860	-59	-23	-15	-15	931	966	975	975
v2	468	778	596	778	565	790	683	790	565	790	683	790	468	778	596	778
v3	-43	40	-27	40	-43	40	-27	40	163	657	525	657	509	563	482	563
v4	-48	-50	1	-48	-48	-50	1	-48	-48	-50	1	-50	-48	-50	1	-48
v5	-27	39	-40	39	451	541	453	541	-27	39	-40	39	451	541	453	541
Total	1281	1773	1505	1784	1853	2283	1970	2183	594	1413	1154	1421	2311	2798	2507	2809

galileo																
	Run 1				*Run 2*				*Run 3*				*Run 4*			
	Tcov -all	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*	*Tcov -all*	*Acov -all*	*Rand -all*	*AHP*
v1	172	691	580	691	-51	461	401	461	-125	715	515	715	172	691	580	691
v2	-115	366	297	366	-114	368	251	368	-114	368	251	368	-115	366	297	366
v3	235	526	381	526	51	242	187	187	-75	452	318	452	235	526	381	526
v4	168	309	380	380	-71	168	309	309	5	56	13	13	5	56	13	13
v5	-3	56	9	-3	667	700	565	667	667	700	565	667	-3	56	9	-3
v6	-115	344	283	344	-115	344	283	344	-49	228	246	228	-40	272	262	-40
v7	-186	216	130	216	-98	224	170	224	-76	285	250	285	-186	216	130	216
v8	-75	379	289	379	-126	364	246	364	-126	364	246	364	-75	379	289	379
v9	-311	204	118	204	-249	224	146	224	-76	469	374	469	-311	204	118	204
v10	-77	456	151	456	-77	456	151	456	-74	405	256	405	-74	405	256	405
v11	-4	575	528	575	-174	579	462	579	-174	579	462	579	-174	579	462	579
v12	-105	148	154	148	-3	136	177	136	-84	228	232	228	-105	148	154	148
v13	-72	112	250	250	-72	112	250	250	-85	-57	120	120	-85	-57	120	120
v14	-86	-251	-249	-86	-83	-216	-124	-83	-122	273	188	-122	-86	-251	-249	-86
v15	-80	211	293	211	-80	211	293	211	-111	-125	64	64	-111	-125	64	64
Total	-654	4342	3594	4657	-595	4373	3767	4697	-619	4940	4100	4835	-953	3465	2886	3582

it outperformed 16 out of 20 cases when we considered the total number of versions that performed best. In the case of *jmeter*, the total cost-benefit of Acov-all was higher than that of AHP, but it did not perform better than AHP when we compared the number of versions that produced the best results by Acov-all and AHP.

VI. Threats to Validity

This section describes the construct, internal and external validity threats to the validity of our study.

Construct Validity: Two issues involve threats to construct validity. (1) We identified four evaluation criteria to apply the AHP method mainly considering the costs

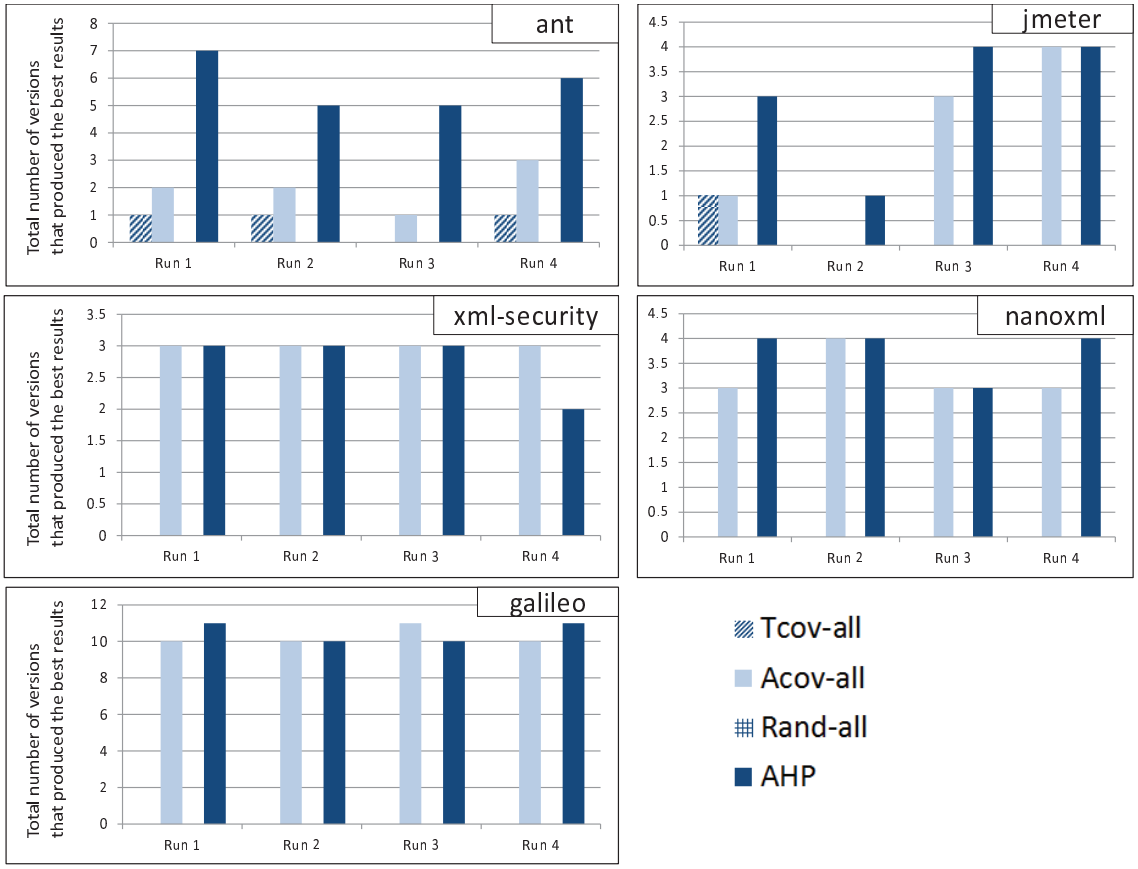


Figure 3. Experiment Results: The Total Number of Versions That Produced the Best Results by Prioritization Testing Strategies

that are associated with test case prioritization techniques. Other evaluation criteria, such as risks for estimated cost-benefit factors, applicability of a technique to a certain type of software artifact, and relevance to the specific testing process, could be considered. (2) The pairwise comparison value in AHP is subject to human judgment (in our case, a graduate student) and thus the results can be biased by personnel's knowledge and experience.

Internal Validity: The inferences we made about the effectiveness of AHP could have been affected by potential faults in our experiment tools. To control this threat, we validated our AHP tool using several examples. Other tools were from SIR [28], and they have been validated through numerous experiments.

External Validity: Three issues limit the generalization of our results. (1) MCDM approach and test case prioritization technique representativeness. In this study, we considered only one type of MCDM approaches and two conventional test case prioritization techniques, so our results cannot be generalized because they are not representative of MCDM approaches and test case prioritization techniques. (2) Object program and mutation fault representativeness. The object programs are of small and medium size. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. We used mutation faults generated by our mutation tool, but there is some evidence that mutation faults can be representative of real faults [30], [34]. Control for these threats can be achieved only through additional studies with wider populations of programs and faults, and different prioritization techniques.

VII. Discussion

We now draw on the results of our analyses, together with additional consideration of our data, to derive practical implications of these results.

ART strategy results: Our results indicate that the prioritization techniques selected by AHP across the entire system lifetime can be more cost-effective than those used by the control approaches with the exception of some cases.

Through the empirical study, we observed the following trends. Overall, the AHP strategy's performance was stable across all programs for all runs, and the Acov-all strategy also produced better results compared to other control strategies, and in some cases (run 4 in *xml-security* and run 3 in *galileo*), it even outperformed the AHP strategy. However, it was not as stable as the AHP strategy. For instance, on *ant* for all runs, the Acov-all strategy was worse than all other

strategies, and on *jmeter*, it was close to the worst case for half of the cases.

In the case of the Rand-all strategy, it was better than the Tcov-all strategy in most cases (except for all cases in *xml-security*). However, since our results for the random technique involve averages of multiple runs, individual random orders may vary widely in performance. The Tcov-all strategy was not worst for all cases, but overall performance is not preferable to others. In particular, in several cases (two runs in *ant*, three runs in *jmeter*, and all runs in *galileo*), it was even worse than the baseline strategy.

Practical implications of the results: So far we have discussed our major findings and the results of our experiment. Now, we discuss practical implications of our results.

From several prior empirical studies of prioritization [14], [15], [32], [33]), we learned that typically prioritization heuristics are more cost-effective compared to control techniques, but we also learned that various factors related to software, its associated artifacts (e.g., program size, test suite size, test suite granularity, and the amount of change between versions), and organization's testing environment could affect the relationships between techniques. Thus, adopting different types of test case prioritization techniques considering such factors is potentially a practical approach for organizations who have time pressure with the product release, due to the constraint budgetary problem and competitive software market.

To our knowledge, our study is the first attempt to investigate the effectiveness of adaptive regression testing strategy. Our proposed strategy produced promising results, and we believe that our empirical methodology and findings from our study provide insights into how such investigation can be performed and what types of MCDM approaches and evaluation criteria can be considered.

VIII. Conclusions and Future Work

In this paper, we have investigated an adaptive regression testing (ART) strategy that utilizes one of the multiple criteria decision making (MCDM) approaches, Analytic Hierarchy Process (AHP), and presented an empirical study assessing the ART strategy. Our results show that our ART strategy can assist researchers and practitioners in choosing cost-effective techniques across system lifetime.

As with all empirical studies, our study also has several limitations as we discussed in Section VI. These limitations can be addressed only through further studies of additional artifacts and regression testing techniques. For future work, we intend to investigate ART strategies further considering several aspects.

First, in this study, we chose the AHP method to implement an ART strategy, but there are many other MCDM approaches available including Weighted Sum Model and modified AHP methods. Thus, the next natural step is to investigate whether different types of approaches help improve ART strategies.

Second, in this study, we used only 4 evaluation criteria, but in order to limit threats to validity as we addressed in Section VI, we intend to investigate ART strategies considering other types of evaluation criteria.

Third, we considered only two test case prioritization heuristics, but we intend to investigate ART strategies that employ other types of prioritization techniques including other regression testing techniques, such as regression test selection techniques. Also, we intend to develop new regression testing techniques so that we can improve our chances of detecting faults under time-constrained situations.

Acknowledgments

This work was supported in part by NSF under Awards CNS-0855106 and CCF-1050343 to North Dakota State University.

References

[1] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the International Symposium on Foundations of Software Engineering*, Nov. 2004.

[2] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of Java programs," in *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2004, pp. 432–448.

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

[4] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, Aug. 2000, pp. 102–112.

[5] A. Srivastava and J. Thiagarajan, " Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis*, Jul. 2002, pp. 97–106.

[6] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 1997, pp. 230–238.

[7] T. L. Saaty, *The Analytic Hierarchy Process*. McGraw-Hill, 1980.

[8] E. Triantaphyllou and K. Baig, "The impact of aggregating benefit and cost criteria in four MCDA methods," *IEEE Transactions on Engineering Management*, vol. 25, no. 2, pp. 213–226, Feb. 2005.

[9] B. R. J. Karlsson, C. Wohlin, "An evaluation of methods for prioritizing softawre requirements," *Information and Software Technology*, vol. 39, pp. 939–947, 1998.

[10] J. E. Steiguer, J. Duberstein, and V. Lopes, "The analytic hierarchy process as a means for integrated watershed management," in *Interagency Conference on Research on the Watersheds*, Oct. 2003, pp. 736–740.

[11] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2009, pp. 201–212.

[12] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[13] S. Yoo and M. Harman, " Regression testing minimisation, selection and prioritisation : A survey," *Software Testing, Verification, and Reliability*, Mar. 2010.

[14] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, Sep. 2010.

[15] 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.

[16] A. Malishevsky, G. Rothermel, and E. S., "Modelig the cost-benefits tradeoffs for regression testing techniques," in *Conf. Softw. Maint.*, Oct. 2002, pp. 204–213.

[17] X. Qu, M. Cohen, and R. G., "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2008, pp. 75–86.

[18] 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.

[19] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, 2004.

[20] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker, "Empirical studies of a prediction model for regression test selection," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 248–263, Mar. 2001.

[21] K. M. A.-S. Al-Harbi, "Application of the AHP in project management," *International Journal of Project Management*, vol. 19, pp. 19–27, Jan. 2001.

[22] R. N. Wabalickis, "Justification of fms with the analytic hierarchy process," *IEEE Transactions on Software Engineering*, vol. 7, pp. 175–182, 1988.

[23] K. E. Cambron and G. W. Evans, "Layout design using the analytic hierarchy process," *Computers and Industrial Engineering*, vol. 20, pp. 221–229, 1991.

[24] T. O. Boucher and E. L. MacStravic, "Multiattribute evaluation within a present value framework and its relation to the analytic hierarchy process," *The Engineering Economist*, pp. 55–71, 1990.

[25] A. Barcusa and G. Montibeller, "Supporting the allocation of software development work in distributed teams with multi-criteria decision analysis," in *Multiple Criteria Decision Making for Engineering*, Jun. 2008, pp. 464–475.

[26] G. E. W. Gavin R. Finnie and D. I. Petkov, "Prioritizing software development productivity factors using the analytic hierarchy process," *Journal of Systems and Software*, vol. 22, pp. 129–139, Aug. 1993.

[27] A. S. Anna Perini, Filippo Ricca, "Tool-supported requirements prioritization: Comparing the AHP and CBRank methods," *Information and Software Technology*, vol. 51, pp. 1021–1032, Jun. 2009.

[28] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *International Journal on Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[29] T. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Comm. ACM*, vol. 31, no. 6, Jun. 1988.

[30] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[31] ——, "Using sensitivity analysis to create simplified economic models for regression testing," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2008, pp. 51–62.

[32] ——, "An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2006.

[33] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Int'l. Conf. Softw. Maint.*, Aug. 1999, pp. 179–188.

[34] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Int'l. Conf. Softw. Eng.*, May 2005, pp. 402–411.