A COMPLEXITY BASED REGRESSION TEST SELECTION STRATEGY

Bouchaib Falah[1], Kenneth Magel[2], and Omar El Ariss[3]

[1]School of Science and Engineering, Al Akhawayn University, Ifrane, Morocco
b.falah@aui.ma
[2]Computer Science Department, North Dakota State University, Fargo, ND, USA
kenneth.magel@ndsu.edu
[3]Computer Scienceand Mathematical Sciences, Pennsylvania State University, Harrisburg, PA, USA
oelariss@psu.edu

ABSTRACT

Software is unequivocally the foremost and indispensable entity in this technologically driven world. Therefore quality assurance, and in particular, software testing is a crucial step in the software development cycle. This paper presents an effective test selection strategy that uses a Spectrum of Complexity Metrics (SCM). Our aim in this paper is to increase the efficiency of the testing process by significantly reducing the number of test cases without having a significant drop in test effectiveness. The strategy makes use of a comprehensive taxonomy of complexity metrics based on the product level (class, method, statement) and its characteristics.We use a series of experiments based on three applications with a significant number of mutants to demonstrate the effectiveness of our selection strategy.For further evaluation, we compareour approach to boundary value analysis. The results show the capability of our approach to detect mutants as well as the seeded errors.

KEYWORDS

Software Testing, Complexity Metrics, Selection Test, Mutant, Regression Testing, Boundary Value Analysis

1. INTRODUCTION

Software testing is the process of determining if a program behaves as expected. It is an intellectually challenging activity aimed at evaluating the capability of a program or system to determine whether or not it meets its requirements. Unfortunately, since testing usually occurs near the end of software development, it is often rushed and frequently not properly carried out.
Software testing is a lengthy, imperfect, and error-prone process. However, it is vital to the success of any product. Furthermore, testing is a critical and essential phase that occurs late in most software development life cycle processes, which often makes it very rushed and frequently not properly carried out.

The complexity of today's software has greatly increased the economic costs of errors. According to a study by the U.S. Department of Commerce National Institute of Standards and Technology (NIST) [1], software bugs cost the U.S. economy an estimated $59.5 billion annually or about 0.6% of the gross domestic product. Additionally, the study found that even if software bugs

DOI : 10.5121/cseij.2012.2501

cannot be removed entirely, more than a third of the costs, or an estimated $22.2 billion, could be saved by earlier, improved testing and more effective detection and removal of software bugs.
In recent years, much attention has been directed toward reducing software costs. To this end, software engineers have attempted to find relationships between the characteristics of programs and the complexity of doing programming tasks or achieving desirable properties in the resulting product such as traceability or security. The aim has been to create measures of software complexity to guide efforts to reduce the software costs.

Even though the body of literature dedicated to the development of testing methods has grown, software complexity metrics have not yet gained wide acceptance as useful guides. Only small experiments have been done in this area and the results provide only weak support for the use of complexity metrics. Nonetheless, many researchers continue to develop and create new metricsto be effective predictors for performing project tasks in less time.

An essential part of any successful testing program is to define and implement specific goals and strategies. During implementation, progress against these goals and strategies must be continuously tracked and measured using various types of testing metrics. Based on the result of these metrics, we can assess the software defects that need to be found during the testing phase and adjust schedules or goals consequently.The testing phases of development are often rushed and incompletely done due to schedule overruns in previous phases. If test cases can be made more effective in finding errors, fewer test cases would be required, resulting in less time and resource expenditure on adequate testing.

In our research, we proposed an approach tolower the cost of and time required for testing and to evaluate software quality by developing a new application tool which measures our suite of complexity metrics and uses it to target test cases. This tool automates the following tasks for Java applications: (a) calculating the values of 20 metrics for any class of the application; (b) computing the values of 9 metrics for any method chosen from that class; (c) finding the values of 4 metrics for any statement within that method; (d) executing all the parts that constitute the application.

The objective is to reduce significantly the number of test cases needed to reveal the presence of a large percentage of the errors that would be discovered through existing approaches. Our goal is to use no more than twenty percent of the test cases to reveal at least sixty percent of the errors. We deal only with unit testing here although we are working to extend our approach to integration and system testing.

Our approach uses a spectrum of complexity metrics at each of three levels of an application (class, method, and statement), that measure all threedimensions of program complexity (size, data, control).Using the entire spectrum of metric values, we target test cases for maximum return. Each metric occupies a different position in a two-dimensional space as shown in Figure 1. The actual metric values in a specific situation form the third dimension.

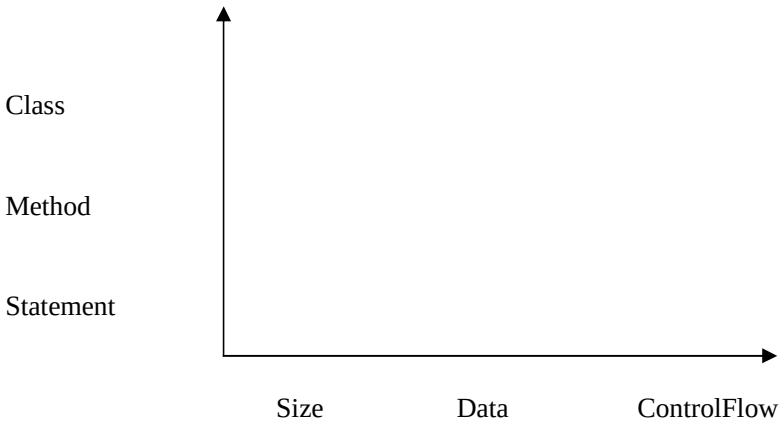

Figure 1. Our two- dimensional metric space

The rest of this work is organized as follows:Section 2 reviews related works. Section 3 describes the spectrum of complexity metrics used to target test cases as well as the test case generation based on this spectrum of metrics. Section 4 describes the empirical study and the results achieved. Section 5 summarizes the contributions of this work and discusses the future work.

2. RELATED WORK

Regression test selection techniques attempt to reduce the cost of software testing by selecting a subset of an existing test suite for execution on a modified program ignoring test cases that cannot or are unlikely to reveal errors caused by or revealed through the specific software changes. To improve the effectiveness of regression test selection, numerous researchers have proposed a variety of test case selection methods. Rothermel et al. [2, 3], Hutchins et al. [4], and Kim and Porter [5] examined different selection test techniques such as minimization, safe, dataflow, control flow, ad hoc and random testing, focusing on their abilities to reduce the cost of testing by decreasing the number of test cases while still detecting the maximum number of defects. However, there is a critical question facing software engineering today: How do we measure the effectiveness of software testing?

Measurement makes interesting characteristics of products more visible and understandable. Appropriate measurement can identify useful patterns present in the product being measured. Many metrics have been invented. Most of them have been defined and then tested only in a limited environment. The most commonly used metrics for software are the number of lines of source code, LOC [6] (a rough measure of size), and McCabeCyclomatic complexity [7] (a rough measure of control flow)].Thomas McCabe's approach was based on the assumption that the complexity of software is related to the number of control paths generated by the code [8].Chidamber and Kemerer [9] proposed a set of complexity metrics that address many principles of object oriented software production to enhance and improve software development and maintenance. However, their metrics applied to only the method and class levels of complexity. In addition, they were evaluated against a wide range of complexity metrics proposed by other software researchers and experienced object oriented software developers. When these metrics are evaluated, small experiments are done to determine whether or not the metrics are effective predictors of how much time would be required to perform some task, such as

documentation or answering questions about the software. Results have been mixed.Nevertheless, industry has adopted these metrics and others because they are better than nothing.

Our work applies a comprehensive suite of complexity metrics to the problem of maximizing the effectiveness of software testing. The frequency with which users of common computer applications encounter serious errors indicates that the industry frequently does an inadequate job of testing before release. However, the problem does not appear to be with the testing process itself. Some software such as that for controlling nuclear reactors is well-tested and has very few errors. Instead, the problem appears to be that the best known methods for testing are too time consuming and complicated to be applied when a project is already running weeks or months behind schedule. Almost always, industry sacrifices best practices in testing to make up some of the time for release.

3. SPECTRUM OF COMPLEXITY METRICS

3.1. Variety of Metrics

Software engineers use measurement throughout the entire life cycle. Developers measure the attributes of software to get some sense of whether the requirements are reliable, consistent and complete, whether the design is of high quality, and whether the code is ready to be tested.Managers measure the characteristics of the product to check when the software will be ready for release and whether the budget will be exceeded. Customers measure aspects of the final product to determine if it meets the requirements, and if its quality attributes are sufficient and effective.

Software metrics usually are considered in one or two of four categories:

- Product: (e.g., lines of code)
- Process: (e.g., test cases produced)
- People: (e.g., inspections participated in)
- Value to the customer: (e.g., requirements completed)

In our work, we concentrated on product metrics as selectors for test cases. Previous work using metrics almost always considered only a small set of metrics which measured only one or two aspects of product complexity.

Our work starts with the development of a comprehensive taxonomy of product metrics. We based this taxonomy on two dimensions:

1. The Scope: The level of the product to which the metric applies;
2. The Kind: The characteristic of product complexity that the metric measures.

The scope of consideration dimension includes the following values: (1) the product's context, including other software and hardware with which the product interacts; (2) the entire product; (3) a single subsystem or layer; (4) a single component; (5) a class; (6) a method; (7) a statement.
For the initial uses of this taxonomy reported in this work, we used only (5), (6), and (7) since they appear to be the most relevant scopes for unit testing. Future work may add (3) and (4) as we consider integration testing and (1) and (2) for system testing.The complexity kind dimension includes the following values: size, control flow, and data. Each of these values in turn has sub-values.

For size, the sub-values are: (a) number of units (e.g. statements); (b) number of interactions (e.g. number of method calls).

For control flow, the sub-values are: (a) number of decisions; (b) nesting depth of decisions. For data, the sub-values are: (a) data usage; (b) data flow.

3.1.1. Metrics at Statement Level

For each executable statement within a method, we had four metrics that emerged from three complexity dimensions:

- Data Dimension: active data values and Data usage values.
- Control Dimension: scope metric.
- Size Dimension: number of operators.

3.1.1.1. Data Dimension

Data complexity metrics can be divided in two different aspects: data flow and data usage. Data flow is based on the idea that changing the value of any variable will affect the values of the variables that depend upon that variable's value. It measures the structural complexity of the program and the behaviour of the data as it interacts with the program. It is a criteria that is based on the flow of data through the program. This criteria is developed to detect errors in data usage and concentrate on the interactions between variable definition and reference.we estimated data flow for each statement in a method by counting how many active data values there are when the method executes. Active data values are counted by determining which variable assignments could still be active when this statement begins execution plus the number of assignments done in this statement. Several testers have chosen testing with data flow because data flow is closely related to Object Oriented cohesion [10, 11, 12].However, data usage is based on the number of data defined in the unit being considered or the number of data related to that unit. We defined data usage for a statement to be the number of variable values used in that statement plus the number of variables assigned new values in that statement.

3.1.1.2. Control Dimension

In our work, we used one control flow measure, the scope metric [13]. For each statement we counted the number of control flow statements (if-then-else, select, do-while, etc.) that contain that statement.

3.1.1.3. Size Dimension

Our size metrics relied on the Halstead Software Science Definition [14]. Halstead's software science is one traditional code complexity measure that approaches the topic of code complexity from a unique perspective. Halstead counted traditional operators, such as + and ||, and punctuation such as semicolon and ( ), where parentheses pair counted as just one single operator. In our work, we counted just traditional operators for simplicity by counting the number of operators used in each statement of the code. Figure 2 shows the metrics we used at the statement level. These four metrics are used as roots to derive other complexity metrics that will be used at the method level and class level

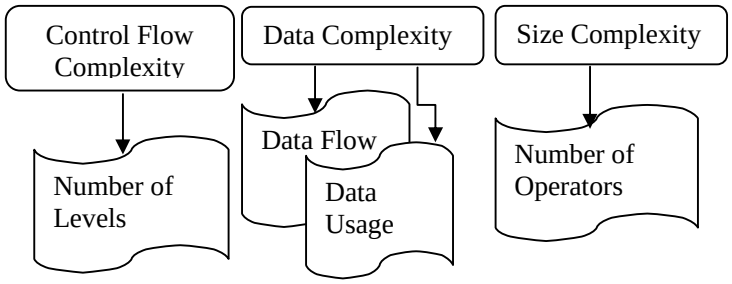

Figure2. Complexity perspective and metrics at statement level

3.1.2. Metrics at Method Level

Since the method constitutes different statements, we used both the sum of each metric for the statements within that method and the maximum value of each metric for the statements within that method. In addition to the sum and the maximum of these metrics, we used another single level metric that counts the number of other methods within the same module (class) that call this method.Figure 3 illustrates the nine metrics used to measure the complexity of a method.

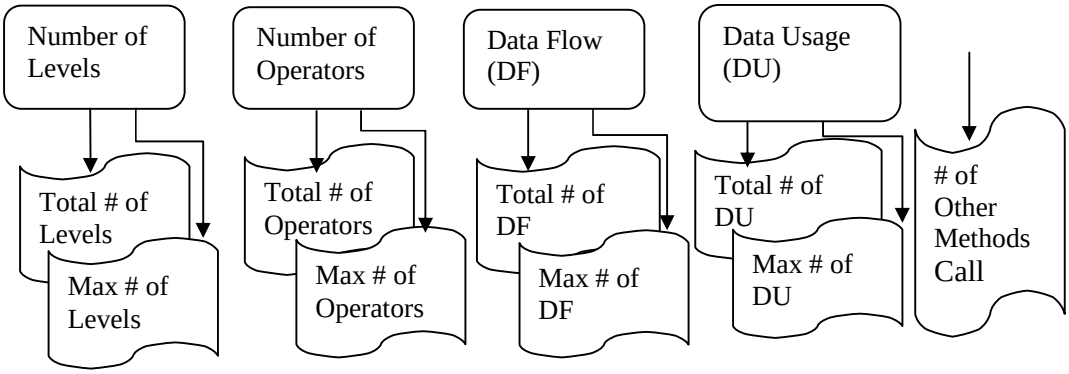

Figure 3. Complexity Metrics at Method Level

3.1.3. Metrics at Class Level

At the class level, we used both the sum of each metric for the methods within the class (e.g. Total of Operators) and the maximum value of each metric for the methods within the class (e.g. Max of Operators, Max of Levels, Max of Tot Levels, Max of Tot DU, Max of Tot DF,…). We then added two additional metrics: the in-out degree of that class, which is the number of methods outside of that class that are called by at least one method in that class, and the number of public members within the class. The public members within a class are defined as the public fields and the public methods defined in that class.

Figure 4summarizes and illustrates how our suite of complexity metrics would be apportioned among our taxonomy dimensions.Each metric of the taxonomy occupies a different position in two-dimensional space.

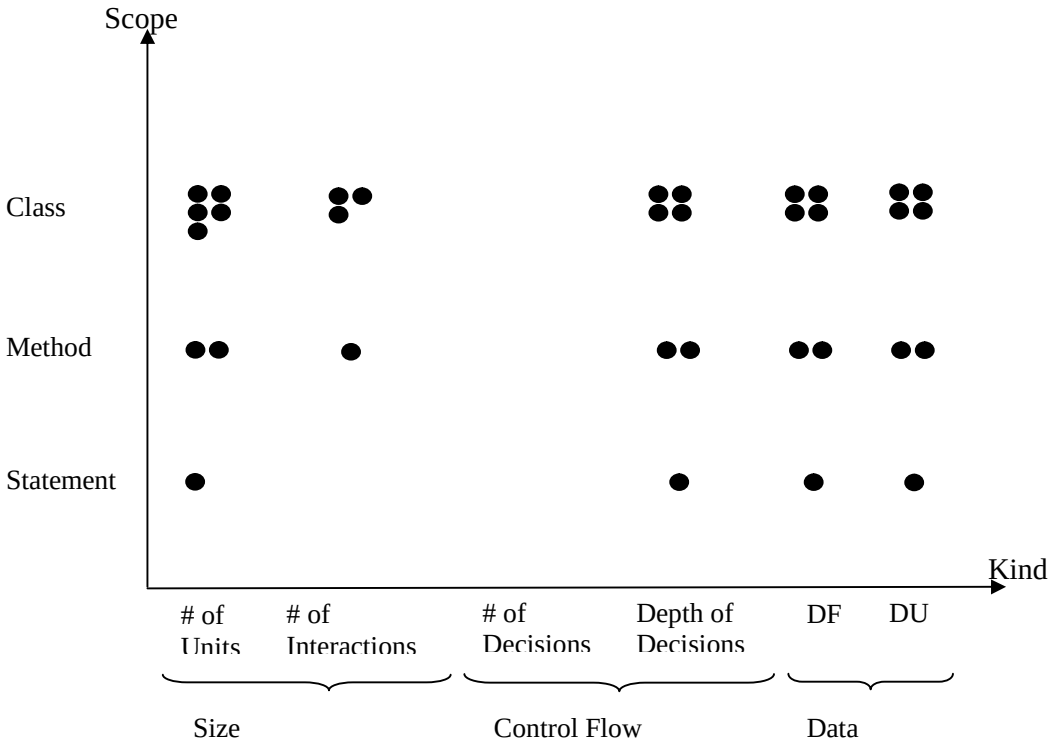

Figure 4. Taxonomy dimensions of our approach

Moreover, we proposed that a combination of size, control flow, and data metrics does better than one dimension. This assumption permitted us to use those values with the objective of finding patterns in the entire set of metrics measurements and identifying sample metric combinations to use in our initial evaluation.

Examples of the combinations we proposed are:

$$Oper\ DF\ Combo = \frac{Max\ of\ Operators + Max\ of\ DF}{Number\ of\ Statements} \quad (1)$$

$$Levels\ DU\ Combo = \frac{Max of Levels + Max of DU}{Number of Statements} \quad (2)$$

$$All\ Combo = \frac{\sum(Max\ of\ Tot\ Operators, Max\ of\ Tot\ Levels, Max\ of\ Tot\ DU, Max\ of\ Tot\ DF)}{Number\ of\ Statements} \quad (3)$$

We do not intend to evaluate all possible combinations. That will be a prohibitive number. Other combinations will be given our attention in future work.

3.2. Test Case Generation Based on a Spectrum of Complexity Metrics

It is not enough for us as software engineers to choose a test strategy. Before testing starts, we must also know the minimum number of test cases needed to satisfy the strategy for a given program. This information helps us to plan our testing, not only in terms of generating data for each test case but also in understanding how much time testing is likely to take and what expenses are necessary to deliver the software.

We have developed algorithms for test case generation based on the values calculated from our combinations of spectrum metrics. The aim of the test case generation is to:

- compare the original program with the seeded programs that resulted from the original program after we employed a third party to insert defects manually into our program;

- compare the original program with the mutants after using MuClipse, , a popular automatic mutation testing tool that is used in Eclipse, to automatically generate these mutants..

If the test cases were able to differentiate a mutant or a seeded program from the original program, then the mutant/seeded program is said to be killed. Otherwise it is still alive.

The evaluation of our approach was done by calculating the ratio of the number of seeded errors to the number of test cases selected and the ratio of the number of mutants distinguished to the number of test cases selected. The fewer the test cases are, the higher the ratios are and, hence, the more effective our testing strategy is. Our goal is to reduce the number of test cases and the cost of testing and, thus, increase the effectiveness of testing. To execute the test cases, we used the procedure pioneered by JUnit of other test case automatic programs.

The test cases are implemented as methods within a new test class associated with the class being tested. An umbrella program then executes each of the test methods against the class being tested and then compares the results to those previously expected.

To reduce the number of test cases, we proposed that our selection testing should be based on our broad spectrum metrics. Our approach is based on the assumption that we should focus on the classes that are more likely to be complex. The targeted classes are more fully testedthan the other classes in the application. These classes are determined by the metrics combination mentioned above. Thus, to test an application, we evaluated the metric values of all classes of the application and concentrated our tests on classes with the highest metric values. These classes are likely to be more complex in the application.

Since our SCM tool evaluates each method within each class, the methods with the highest metrics would be our target.

4. EMPERICAL STUDY

As a basis for our empirical study, we chose three applications downloaded from sourceforge.net, a popular repository for open source projects: CruiseControl, Blackjack, and CoffeeMaker.

4.1. Our Approach

We have developed algorithms for test case generation based on the values resulting from our metric combinations (1), (2), and (3) given above.These test cases were created using unit testing, then were run on our programs after these programs have been modified by an automatic mutation testing tool such as MuClipse and independently by a 3rd party manually.

4.2. Mutation Testing and Error Seeding

We used two different methods to evaluate testing effectiveness: mutation testing; and error seeding by an independent person. For each class, all methods were tested, but our focus was on the most complex methods. We started with the creation of one test case for each method. Then we ran these test cases on the modified programs and calculated:

- The score of mutants: the number of killed mutants divided by the total number of mutants.
- The detection rate of seeded errors: the number of the detected seeded errors divided by the total number of all seeded errors.

When the score of mutants or the detection rate of the seeded errors was too low, we increased the number of test cases for the complex methods until we got a better score.

4.3. Evaluation of Our Approach

To assess our approach and show its effectiveness, we compared it with other approaches that have been used in the software engineering discipline. Since Boundary Value Analysis (BVA) is the mostwidely practiced form of functional testing, we created BVA test cases and ran them on our modified programs in the same way we did for our test cases. In creating BVA test cases, we concentrated our testing efforts on the limits of the boundary of each method within each class subject to test.

The comparison was based on:

- Mutants, by comparing the mutant score divided by the number of test cases.
- Seeded programs, by comparing the detection rate divided by the number of test cases.

Since our regression selection testing approach used three combinations of metrics, the comparison involved evaluating our approach against the boundary value analysis for each combination metric.

In the rest of this paper, we assume that m and s are two real numbers such that:

$$m = \frac{Mutants\ killed\ score}{Number\ of\ test\ cases}$$

$$s = \frac{Percentage\ of\ seeded\ errors\ detected}{Number\ of\ test\ cases}$$

4.3.1. Mutation Testing Technique

Our approach used the regression selection tests based on the values of the three combination metrics for each class within each application. Tables 1, 2, and 3 provide statistics for every class tested. For each testing class and for each approach, we list:

- The number of test cases created for each class,
- The mutant score, which is the number of mutants killed divided by the total number of mutants
- The number of mutants divided by the total number of test cases.

Table 1. Comparison on mutants for operators DF Combo metric

Application	Class Name	Our Approach			Boundary Value Analysis		
		# of test	Mutant	m	# of	Mutant	m
Blackjack	Dealer	3	100%	33	6	100%	17
	Hand	4	100%	25	11	100%	9
	User	6	100%	17	14	100%	7
CoffeeMaker	CoffeeMake	12	100%	8	10	21%	2
	Inventory	16	66%	4	22	43%	2
CruiseControl	CarSimulato	13	100%	8	39	100%	3
	SpeedContr	8	100%	13	19	100%	5

Table 2. Comparison on mutant programs for levels DU Combo metric

Application	Class Name	Our Approach			Boundary Value Analysis		
		# of	Mutant	m	# of	Mutant	m
Blackjack	Dealer	3	100%	33	6	100%	177
	Hand	5	100%	25	11	100%	9
	LogicalFacade	18	76%	17	32	52%	7
CoffeeMaker	CoffeeMaker	10	100%	8	10	21%	2
	Inventory	20	61%	4	22	43%	2
CruiseControl	CarSimulator	12	100%	8	39	100%	3
	Controller	10	100%	13	45	100%	5

Table 3. Comparison on mutant programs for All Combo metric

Application	Class Name	Our Approach			Boundary Value Analysis		
		# of tests	Mutant Score	m	# of tests	Mutant Score	m
Blackjack	Dealer	3	100%	33	6	100%	17
	Hand	5	100%	20	11	100%	9
	User	6	100%	17	14	100%	7
CoffeeMaker	CoffeeMaker	9	100%	11	10	21%	2
	Inventory	16	66%	4	22	43%	2
CruiseControl	CarSimulator	13	100%	8	39	100%	3
	CruiseControl	4	100%	25	16	100%	6

Based on the statistical results in these tables, we observed that the value of m in our approach is higher than the one in BVA for all testing classes.

Based on our observation, there are many different data inputs on each method of this class, which makes the number of test cases using BVA technique greater, and thus the value of s was lower compared to the one in our approach.

As a summary, our empirical study shows that, using either mutation testing technique or the error seeding testing technique, the rate of mutant score and the rate of detected errors with respect to the number of test cases is a good indicator of the efficiency of our testing approach against the boundary value analysis approach.

4.3.2. Seeded Error Technique

Error Seeding was done by a third party who intentionally inserted faults in the source code of each of the classes used in this research experimentfor the purpose of monitoring the rate of detection. These faults were inserted according to the top errors usually made by Java programmers [15].

In this technique, we used the same test cases that were used in the mutation testing but with a smaller number for some. Then, we determined the rate of detection obtained for each class by dividing the detected seeded errors by the total number of seeded errors. The experimental results, obtained based on each combination metric, are shown in table 4, 5, and 6.

We used a smaller number of test cases in the error seeding technique than in the mutation testing technique. This was due to the fact that we have a smaller number of seeded errors compared to the number of automatic mutants.

Based on the experimental results in Table 4 and 5, we observed that the values of s for all tested classes in our approach are much higher than for BVA.

Table 4. Comparison on seeded programs for operators DF Combo metric

Application	Class Name	Our Approach			Boundary value Analysis		
		# of tests	Detection Rate	s	# of tests	Detection Rate	s
Blackjack	Dealer	2	100%	50	6	100%	17
	Hand	4	100%	25	11	100%	9
	User	6	100%	17	14	100%	7
CoffeeMaker	CoffeeMaker	9	100%	11	11	66%	6
	Inventory	11	100%	9	22	100%	5
CruiseControl	CarSimulator	12	100%	8	39	100%	3
	SpeedControl	8	100%	13	19	100%	5

Table 5. Comparison on seeded programs for levels DU Combo metric

Application	Class Name	Our Approach			Boundary value Analysis		
		# of tests	Detection Rate	s	# of tests	Detection Rate	s
Blackjack	Dealer	2	100%	50	6	100%	17
	Hand	4	100%	25	11	100%	9
	LogicalFacade	18	100%	6	32	100%	3
CoffeeMaker	CoffeeMaker	9	100%	11	11	66%	6
	Inventory	14	100%	7	22	100%	5
CruiseControl	CarSimulator	12	100%	8	39	100%	3
	Controller	10	100%	10	45	100%	2

Table 6. Comparison on seeded programs for All Combo metric

Application	Class Name	Our Approach			Boundary value Analysis		
		# of tests	Detection Rate	s	# of tests	Detection Rate	s
Blackjack	Dealer	2	100%	50	6	100%	17
	Hand	4	100%	25	11	100%	9
	User	6	100%	17	14	100%	7
CoffeeMaker	CoffeeMaker	9	100%	11	11	66%	6
	Inventory	11	100%	9	22	100%	5
CruiseControl	CarSimulator	12	100%	8	39	100%	3
	CruiseControl	4	100%	25	16	100%	6

The results of Table 6 show that even though the detection rates for all the classes, except the "CoffeeMaker" class, in both approaches are the same, the values of s in our approach are much higher than for BVA, except for the "Inventory" class, where the value of s is close to the one for BVA. This closeness of the value of s is attributable to the detection rates being the same and the number of test cases being close.

5. CONCLUSION AND FUTURE WORK

This paper presented a methodology to lower the cost of software and evaluate its quality. This strategy consisted of creating measures of software complexity that can be used to target test cases where they would be most effective. We started with the development of a comprehensive taxonomy of product metrics. This taxonomy was based on the metric dimension (product level) and the kind dimension (product complexity characteristic). The result was an effective subset of test cases for unit testing

One contribution of this paper is that we used at least one sub-category from each complexity kind dimension value. Another contribution of this research was the use of summation and maximum to build larger scope metrics from smaller scope metrics. The entire spectrum of complexity metrics were applied to three simple Java applications. Based on the results of the values of these metrics, we assessed the software defects that needed to be found during the testing phase and consequently adjusted the schedules or goals. We achieved the reduction of both cost and time of software by the use of a selection test that was based on the set of all of these complexity metrics.

This paper also presented Spectrum of Complexity Metrics (SCM), an implementation program tool, which automated the following:

- the values of twenty metrics for any class of Java application
- the values of nine metrics for any method chosen from that class
- the values of four metrics for any statement within that method
- the execution of all the parts that constitute the Java application.

To assess the effectiveness of our test case selection strategy, we did the following before we ran our test cases on the modified programs:

1. seeded our targeted classes with automatic defects using MuJava, a plug in for Eclipse
2. manually seeded targeted classes with errors, this was done independently by a third party

The effectiveness of our test case generation was evaluated based on both the values of the mutant score and the detection rate. In addition, we compared our approach with the Boundary Value Analysis (BVA) using mutation testing technique and error seeding technique. The results showed that our approach was more efficient than other approaches and was effective in reducing the cost and time of software. Since the statement, the method, and the class appear to be the most relevant scopes for unit testing, they were the only scope dimension values used in this paper. A natural direction for future work is to produce a comprehensive taxonomy from other kinds of metrics.

REFERENCES

[1] Michael Newman (June 2002) "NIST New Releases: NIST Assesses Technical Needs of Industry to Improve Software-Testing", http://www.abeacha.com/NIST_press_release_bugs_cost.htm

[2] T. L Graves, M. J. Harrold, J. Kim, A. Porter & G. Rothermel(2001) "An Empirical Study of Regression Test Selection Techniques", ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, pp184-208.

[3] G. Rothermel&M. J. Harrold (1997) "A safe, efficient regression test selection technique", ACM Transactions on Software Engineering and Methodology, Vol. 6, pp173 – 210.

[4] M. Hutchins, H. Foster, T. Goradia&T. Ostrand (1994) "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria", In Proc of the 22ndInternational Conference on Software Engineering, pp191-200, IEEE.

[5] Jun-Ming Kim & Adam Porter (2002) "A history-based test prioritization technique for regression testing in resource constrained environments", International Conference on Software Engineering, Proceedings of the 24th International Conference.

[6] F. Damereu (1964) "A Technique for Computer Detection and Correction of Spelling Errors", Communications Of the ACM, Vol. 7 Issue 3.

[7] Thomas McCabe and Charles Butler (1989) "Design Complexity Measurement and Testing", Communications of the ACM, Vol. 32, Issue 12.

[8] Thomas J. McCabe (1976) "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4.

[9] R.M. Hierons (2006) "Avoiding Coincidental Correctness in Boundary Value Analysis",Transactions on Software Engineering and Methodology (TOSEM).

[10] S. R. Chidamber& C.F. Keremer (1994) "A Metric Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No 6,pp476-493.

[11] Frank Tsui, Orlando Karam& Stanley Iriele (2008) "A Test Complexity Metric Based on Dataflow Testing Technique", Internal Report supported and funded from PSLSAMP grant.

[12] Henderson-Sellers, B (1996) "Object-Oriented Metrics: Measures of complexity", Prentice Hall.

[13] G. Rothermely, S. Elbaumz, A. Malishevskyy, P. Kallakuriz&X. Qiuy (2003) "On Test Suite Composition and Cost-Effective Regression Testing".

[14] Halstead. M. H (1977) "Elements of Software Science. Elsevier North-Holland", New York.

[15] S. Kim, J. Clark & J. McDermid (2000)"Class mutation: Mutation Testing for Object-Oriented Programs", OOSS: Object- Oriented Software Systems.