Novel Approach to Automated Test Data Generation for AOP

Anuranjan Misra, Raghav Mehra., Mayank Singh, Jugnesh Kumar, and Shailendra Mishra

Abstract—**Aspect oriented programming is a new programming paradigm. AOP is based on object oriented programming. Most of the researchers target this new paradigm towards the programming not for testing. Testing of aspect oriented programs is an emerging field of research as a very few research work is going on currently on ASP.**

In this paper, we investigate a new way of testing aspect oriented programs. Here we propose a framework of automated test data generation for evolutionary testing on AOP. On the basis of generated data we will compare evolutionary testing with random testing in terms of effort reduction and improvement of test effectiveness. We will justify our comparison with the help of empirical study on AspectJ programs.

Index Terms—**Aspect oriented programming, testing AOP, debugging, and Search based optimization techniques.**

I. Introduction

In software development life cycle, testing is important part. The IEEE definition of testing is "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results." The quality of any software product is checked through the testing. More than half of budget of a software project spend on testing even though, it is not guaranteeing the correctness of software. There has been a high level of interest to automate the testing process in software development. To assuring the quality of aspect oriented projects, testing is the only process.

The process of automated software testing requires an approach to select the test case. The main aim of testing is to cover the programming features. Code coverage is used to measure the extent of software code to be tested. Structural testing involves branch and path coverage testing, which is based on measuring the software code. Branch coverage is widely used testing techniques and it is the basis of several industry standards because it is not an extremely strict coverage criterion [6, 13]. These standards are used without any concern for the programming paradigm adopted. Manual test data generation for achieving code coverage is expensive, error prone. Branch coverage is the most widely used technique, so it gains a lot of attention from the testing researchers [14, 28]. Evolutionary testing technique has been very effective at automated test data generation for branch coverage [19, 28].

Automated test data generation is easier for object oriented programs than aspect oriented programs. In this paper we investigate two testing techniques i.e. evolutionary testing and random testing, for aspect oriented programs. These two testing techniques are not new for object oriented, but there is less empirical study available for aspect oriented programs. There has been much interest in advanced automated test data generation techniques for procedural or object oriented programs but none of these techniques has been applied to the AOP paradigm. Because we don't have sufficient previous data about this, so we are unable to measure that how well these techniques can be applied to AOP.

In this paper we give an empirical study for automated test data generation for evolutionary testing. This paper provides a framework for automated testing of aspect oriented programs. Here we also presents a software tool that will help to automate the process of testing aspect oriented software and identify the ways to reducing the effort and increasing the effectiveness of testing. Here we also give the empirical result to show the reduction of test inputs for evolutionary testing of aspect oriented programs. With the help of reduction test input data, we will show the effectiveness of evolutionary testing on AOP. This paper also gives the comparison study of evolutionary and random testing on the basis of automatic generated test data. Reduction of test input is done through the removing of irrelevant branch parameters. Slicing removes any part of the program that cannot influence the semantics of interest in any way. Our empirical study uses only AspectJ language programs.

II. Difference between Random Testing and Evolutionary Testing

Random testing is searching based software testing technique in which test data has been chosen randomly. Random testing helps to cover many structural targets as, usually many sufficient input data sets exist which can be selected to execute those structures in code [10, 14, 21]. Evolutionary testing is also a search based software testing approach based on the theory of evolution. Theoretically it was proven that evolutionary testing achieves better performance. In this paper, we will prove this theoretical

Manuscript received June 2, 2011.

Anuranjan Misra is working as Professor and Dean Academics at Bhagwant Institute of Technology, Ghaziabad, India.

Raghav Mehra is working as Associate Professor, at Bhagwant Institute of Technology, Muzaffarnagar, India.

Mayank Singh is working as Assistant Professor, Computer Science Department, J.P. Institute of Engineering & Technology, Meerut , India.

Jugnesh Kumar is working as Assistant Professor at Manav Rachna College of Engineering, Faridabad, India.

Shailendra Mishra is working as Professor and Head of Department of Computer science and Engineering at kec dwarahat uttarakhand

concept with empirical study using AspectJ programs.

III. Approach

In this paper, we present a framework for automated generation of test data for aspect oriented programs. This framework is based on existing framework for object oriented programs. The main objective of this framework is to generate test data to achieve aspectual branch coverage and calculate total effort. On the basis of this framework, we implement a testing tool which integrates both random testing and evolutionary testing. We use AspectJ programs to implement this framework. The generated test data is only unit tests for the base code with respect to gcaspect code. This generated unit test data can also be used for integration testing. Here our main concern only the aspectual branches instead of all the branches.

In this framework, we first have AspectJ code which will be converted into plain java code. Then we use slicing to identify the aspectual branches only. These aspectual branches are our main aim to test in evolutionary and random testing of AOP. Aspectual branch includes predicates and the methods in aspects. After finding the relevant aspectual branches now we have to identify the relevant parameters of the methods of the base classes. Because in the branch all the parameters are not relevant to test, so the irrelevant parameters are identified and removed from the base class. With the help of this step, we reduce the data for testing. Now the actual evolutionary testing is performed on the relevant parameters of aspectual branches and generates the test data. On the basis of generated test data, the coverage of aspectual branches is calculated and measures the effort required to generate this.

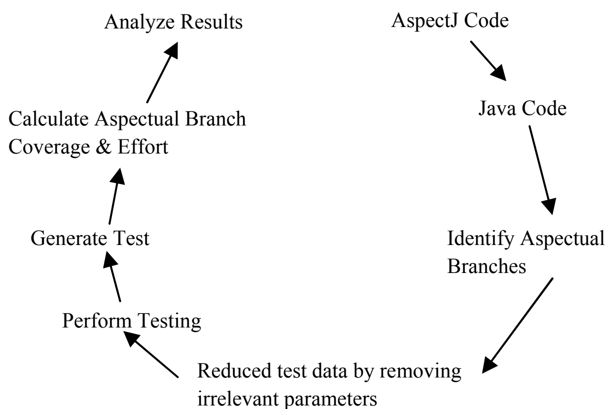

Fig. 1. Framework for automated testing of AOP

To convert code from AspectJ to Java code, AspectJ Compiler 1.0.6 has been used. To identify aspectual branches we use an aspect branch coverage measuring tool "AspectMeasure" which is developed in Java. This framework has been implemented with a software tool which can test Java programs using random and evolutionary testing techniques. To test each branch we use JUnit based test suit, which generate at least one test case for each identifies branch. Finally we measure the coverage of aspectual branches. The effort calculations for the testing process are output based on the testing techniques used.

For evolutionary testing, effort is calculated by in terms of runtime and number of evaluations. For random testing it is calculated using number of generations for random testing. As the effort for evolutionary and random testing is calculated using the same way, the results are directly comparable.

A. Input domain reduction framework for evolutionary testing

The input domain reduction techniques [17, 18] was introduced for constraint based testing. It typically involves simplifying constraints using various techniques and generating random inputs for the variable with the smallest domain. The process is repeated until the target structural entry such as a branch has been parameters.

This framework has been implemented as an extension to the framework proposed above. This design uses evolutionary testing technique to test aspectual branches where test data is reduced through the reduction of irrelevant parameters. These reduced test inputs are used to test the target method containing the branch.

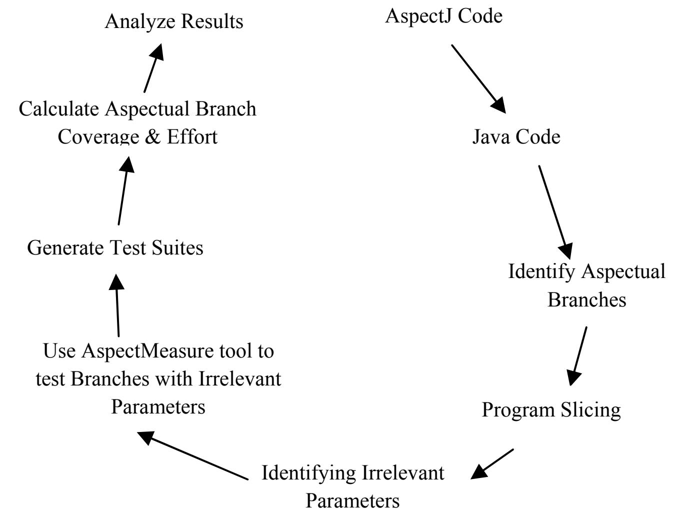

Fig. 2. Framework for reduction of test inputs for ET of AOP

With the help of slicing the irrelevant parameters has been identified for each aspectual branch. The occurrence of each parameter is checked within the slice to determine its relevancy. If a parameter name or its type does not appear within the slice, then it is considered as an irrelevant parameter. After slicing and finding the relevant parameters for test, a new java code is produced for each branch where reduced input test data is possible. Then we generate tests for that branch and the resulting aspectual coverage and effort in terms of number of fitness evolutions is recorded. Original program is tested, the effort and coverage is measures. Then slicing data is tested and measured to compare this data with the original measured data. This result will tell us that how much reduction is possible in effort and improvement in coverage of branches.

IV. Slicing

Slicing is a static analysis technique that helps to create a

reduced version of a program by placing its attention on selected area of semantics. The process removes any part of the program that cannot influence the semantics of interest in any way. Program slicing can be applied to the field of software testing, measurement, debugging or used to better understanding the internal working of a program [12, 13].

The reduced version of the program is called slice and the semantics of interest is known as slice criterion. Based on the slice criterion, it is possible to produce backward or forward slices. Backward slice consists of the set of statements that can influence the slice criterion based on data or control flow. Forward slice contains the set of statement that are control or data dependent on the slice criterion, which includes any statement that can be affected by the slice criterion. In this paper we use backward slices of Java programs [12, 13].

V. Evolutionary Testing

Evolutionary testing is a popular testing technique which is based on theory of evolution. Every generation produced by applying genetic operators to individually which imitate the mating and transformation of natural generics. As the generations increase, the population contains more individuals with high evolution function. The procedure stops when an adequate amount of fitness is has been achieved or the maximum number of generations have been reached. This method of testing has been found to achieve better performance than random testing since it concentrates the search towards finding test data with high fitness values. [33]. Typically approximation level and local distances are used in combination for fitness calculation of individual test data. For branch coverage, a fitness value closer to 0 is desired as a fitness value of "0" means that the branch has been covered. [33]

Example: Fitness calculation for branch coverage-

Suppose, X = 10 and Y = 5.

Target Predicate: if (X=Y)

Local Distance = |X-Y| = 5

Approximation Level = 2 (Suppose)

Fitness = Local Distance + Approximation Level = 5+2 = 7.

VI. Implementation

Firstly Aspectj program under test is compiling using the AspectJ compiler component. To compile the AspectJ files we use ajc compiler. Then the AspectJ code is converted into equivalent Java code using the code convertor component. The resultant Java code is compiled using Java compiler component. The branch identifier component is used to find aspectual branches from the Java code. Based on user preference, the program then forward the identified branches either to test goal generator for random or evolutionary testing or to code slicer for evolutionary testing of aspectual branches with input domain reduction.

If the former route is choosing, then all aspectual branches are sending to the test goal generator for testing. If the later route is chosen, backward slicing is preformed using the code slicer component. On each aspectual branch and the resulting slices and previously identified branches are passed on to the code parser component. The code parser component identifies the branches where input domain reduction by removing irrelevant parameters is possible. A new version of code for each of these identified branches is generated using the code transformer component and the branches are then forwarded to test goal generator for testing transformed versions of the code with reduced parameters.

VII. Experimental Setup

In the empirical study we use a suite of 10 programs written in AspectJ to apply the proposed approach. The following table gives the details of these programs that showing the program name, the line of code of the complete program, the base classes used to drive the aspects under test together with the aspects, the number of aspectual branches which include both branches from predicates in aspects and methods in aspects and the number of aspectual branches from predicates in aspects. Only the aspectual branches from predicates are used to conduct domain reduction in the empirical studies.

TABLE1. Programs Which Are Used in Study

Program	Whole Program LOC	Test driver LOC	Aspectual Branches /Targets	Aspectual Branches from Predicates
Hello	86	33	3	0
Figure	325	147	1	0
NullCheck	134	134	6	4
QuickSort	204	127	4	0
Queue	429	429	12	12
DCM	375	375	86	82
ProdLine	907	907	21	8
PushCount	137	119	1	0
LawOfDemeter	1041	185	107	107
NonNegative	116	94	5	4

In the evolution of aspectra, xie and zhao were used these programs [43]. These programs also include one aspect oriented design pattern implementation by hannemann and Kiczales [16]. These AspectJ programs are the benchmarks which include exception handling, updating and filtering. In the empirical study we compare random testing with evolutionary testing for aspect oriented programs. To get the correct data, we had 20 trials. Each trail is applied on both random and evolutionary testing techniques.

VIII. Empirical Study

The main goal of this empirical study is to find the effectiveness of evolutionary testing technique with comparison to random testing of aspect oriented programs. Till date, random testing has been heavily used to test aspects but for evolutionary testing we don't have any test data.

A. Metrics & Measures

Test adequacy criterions are used to measure how much of the program has been testing. Aspectual branch coverage is considered as the test adequacy criterion in this study as it

is the industry standard for test measurement. The effort for testing is measured using different metrics for evolutionary and random testing. In evolutionary testing, the number of fitness evaluations required to cover a branch is considered as standard measurement for effort. However, random testing uses the number of generation as the measurement for effort. In this comparison, the number of fitness evaluations for evolutionary testing and the number of generations for random testing are directly comparable. The upper bound set for number of evaluations for evolutionary testing and the number of generations for random testing has been set to 10,000.

The 10 subjects introduced previously have been used to conduct experiment in the study. The experiments involves testing aspect oriented programs using evolutionary testing techniques to compare and analyze their result in terms of effort taken for testing and code coverage. The table 2 presents the list of programs and their classes that was possible to be tested. The table also states the number of aspectual branches used as targets for testing. The aspectual branches include branches of all predicates, as well as pointcut branches which translate to covering AOP related methods after weaving.

In table column 1 represents the numbering of classes. Column 2 represents the program under test. Column 3 represents the name of classes under test. Column 4 represents the number of aspectual branches that were used as targets for testing.

TABLE2. TEST OBJECTS CLESSES WITH ASPECTUAL BRANCHES

No	Program	Class	Aspectual Branches
1	Hello	HelloAspetcs	3
2	Figure	DisplayUpdating	1
3	NullCheck	Stack6	6
4	QuickSort	Stats	4
5	Queue	QuesueStateAspect	12
6	DCM	ClassRelationship	2
7	DCM	Metrics	36
8	DCM	Stack4	48
9	ProdLine	CC	2
10	ProdLine	Cycle	2
11	ProdLine	DFS	4
12	ProdLine	MSTKruskal	4
13	ProdLine	MSTPrim	4
14	ProdLine	Number	2
15	ProdLine	Weighted	3
16	PushCount	StackOrig	6
17	LawOfDemeter	Percflow	31
18	LawOfDemeter	Pertarget	76
19	NonNegative	NonNegative	5

The result presented in the above table indicates that a total of 19 classes were possible to be tested from all 10 programs. It is worth monitoring that tool cannot instrument all classes that relate to abstract classes, interfaces or contain static global variables of primitive types. The total number of aspectual branches tested is 251. All branches have been considered in this study regardless of whether they were covered during the test or not.

Here we represent a graph that shows the result of effort comparison between evolutionary and random testing. The x-axis in the graph represents all classes that have been testing and y-axis presents the percentage reduction in effort achieved by evolutionary testing in comparison to random testing.

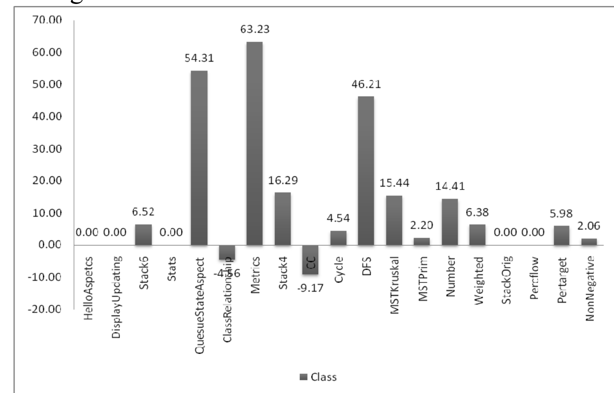

Fig. 3. Effort Reduction in Classes

The results in the graph indicate that 2 out of 19 classes has an increase in effort in evolutionary testing, 5 out of 19 classes took the same effort and remaining 12 branches had reduction in effort when compared to random testing.

Further investigation revealed the fact that the classes which had increase in effort for testing consisted of trivial branches. Theoretically, evolutionary testing should take less effort for covering branches when compared to random testing, but in the case of these two classes that was not the case. A thorough investigation revealed that evolutionary testing took more average effort due to random spikes in the number of evolutions. This is possible evolutionary algorithms have random test data generation mechanism at the heart of it.

The blow graph presents the improvement in coverage after using evolutionary and random testing technique on the 10 programs under test. The x-axis represents each program and the y-axis represents the improvement in branch coverage as a result of using evolutionary testing. The improvement in coverage is calculated using the following formula-

Coverage Improvement = Evolutionary Testing Coverage – Random Testing Coverage

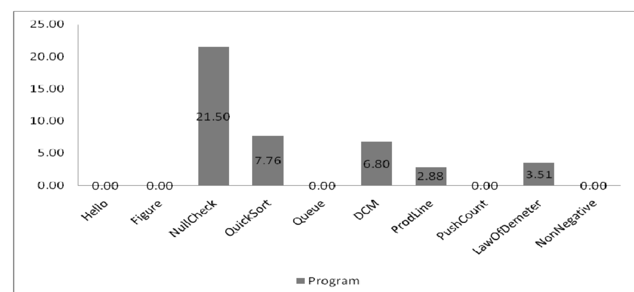

Fig. 4. Coverage Improvement in Tested Programs

From the graph it is observed that 4 out of 10 programs achieved the same branch coverage with evolutionary and random testing. The remaining 6 programs obtained better branch coverage with evolutionary testing. An interesting observation is also made when the results of branch coverage improvement and effort reduction are compared. It is seen that all 5 programs which had an improvement in branch coverage also has a reduction in effort using evolutionary testing.

So we can conclude this study is that Evolutionary testing does not only achieve better branch coverage than random

testing, it also does it with less effort. This study provides evidence that evolutionary testing is a better technique for testing aspect oriented programs in comparison to random testing.

IX. THREATS TO VALIDITY

The threat to external validity primarily includes the degree to which the subject programs and testing techniques under study are representative of true practice. The AspectJ benchmarks are collected from the web and reused benchmarks used in the literature in testing and analyzing aspect oriented programming. Another threat is that the random testing test data has not been generated by us. We have used previously tested data by the researchers, so the validity of random testing data is under doubt.

A potential source of bias exists of a relatively large number of test subjects and branches were not used in the experiments. Another source of bias can be the result of not using a wide variety of programs. These threats were overcome by using as many test subjects as possible obtained form a variety of source. For the experiment concerning domain reduction, all test subjects which was possible to be tested have been used.

X. RELATED WORK

Quite a few approaches have been proposed for testing aspect oriented programs, which includes model checking, data flow and state based testing [4, 6].

There exists neither previous approach to optimization of test data generation nor empirical result on advanced test data generation (beyond random testing) for AOP. This current lack of AOP test automation progress and the associated empirical paucity poses barriers to increased uptake and practical application of AOP techniques [9].

In 2002, model checking was first presented by G. Denaro and M.Monga, to verify various aspect properties appropriate for formal verification. Later a similar approach based on three valued model was proposed by H. Li, S. krishnamurthi and K. Fisler, which verified the features and interections as a result of weaving aspect oriented programs [2, 6].

State based testing approach for aspect oriented programs where introduced in 2005 by D. Xu, W. Xu and K. Nygard. This involved using aspectual state model to record the effects of aspects on the state models of classes [4].

JamlUnit was proposed by C.V.Videira and T. C. Ngo, as an aspect oriented extension of the Java unit testing framework called JUnit. It was specifically developed for Java Aspect Markup Language where aspects are represented using Java base classes and XML binder [5, 8].

Dr. T. Xie introduces the Aspetcra framework for aiding in the automated testing of aspect oriented programs to reduce manual effort in testing [6].

Other related work on the general area of testing aspect oriented programs include fault model for AOP [3,4,5,13], which could potentially be used to help assess the quality of the test data generated by our approach in addition to the aspectual branch coverage being used currently [45].

XI. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel approach to automated test data generation for AOP. This approach is based on evolutionary testing, which uses search based optimization to target hard to cover branches. This paper proposed a framework for testing aspect oriented programs automatically using existing object oriented testing tools, where aspectual branches of the program are identified and tested.

Another framework has been proposed based on this framework which enables to reduce the size of the input domain for evolutionary testing. The result of empirical study on several hundred search problems drawn from 10 AOP benchmark programs show that the evolutionary approach is capable of producing significantly better results than the current state of the art.

In future work we can extend the implementation tools for conducting more experiments to get more results on automated testing of aspect oriented programs. Our future plan is to develop other more advanced test data generation techniques such as mutation testing techniques. To apply mutation techniques first we have to identify the mutant operators for aspect oriented programs, then to develop a tool for empirical study about mutation testing. In this paper, empirical study based on branch coverage, so our future plan is to achieve other type of coverage in AOP systems like data flow coverage between aspect and base code.

REFERENCES

[1] AspectJ compiler 1.2,. http://eclipse.org/aspectj/. (Accessed on 31st Jan, 2011)

[2] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.

[3] K. Arnold, J. Gosling, and D. Holmes. The Java Programming Language. Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] B. Beizer. Software Testing Techniques. International Thomson Computer Press, 1990.

[5] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. Commun. ACM, 44(10):51–57, 2001.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In Proc. International Symposium on Software Testing and Analysis, pages 123–133, 2002.

[7] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In Proc. the International Symposium on Software Testing and Analysis, pages 39–48. ACM Press, 2000.

[8] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. Software: Practice and Experience, 34:1025–1050, 2004.

[9] Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. http://research.microsoft.com/fse/asml/doc/AsmLTester.html.

[10] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In Proc. International Symposium on Software Testing and Analysis, pages 112–122, 2002.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In Proc. 3rd International Conference on Aspect-Oriented Software Development, pages 26–35, 2004.

[12] JUnit, 2003. http://www.junit.org.

[13] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 553–568, April 2003.

[14] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Proc. 11th European Conference on Object-Oriented Programming, pages 220–242. 1997.

[15] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. Commun. ACM,44(10):39–41, 2001.

[16] Parasoft. Jtest manuals version 4.5. Online manual, April, 2003. http://www.parasoft.com/.

[17] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In Proc. 12th International Symposium on the Foundations of Software Engineering, pages 147–158, 2004.

[18] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In Proc. the Workshop on Program Analysis for Software Tools and Engineering (PASTE'04), pages 49–51, June 2004.

[19] A. L. Souter, D. Shepherd, and L. L. Pollock. Testing with respect to concerns. In Proc. International Conference on Software Maintenance, page 54, 2003.

[20] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In Proc. 21st International Conference on Software Engineering, pages 107–119, 1999.

[21] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In Proc. 15th IEEE International Conference on Automated Software Engineering (ASE), pages 3–12, 2000.

[22] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 97–107, 2004.

[23] T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle,WA, Jan. 2004.

[24] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In Proc. 19th IEEE International Conference on Automated Software Engineering, pages 196–205, Sept. 2004.

[25] T. Xie, D. Marinov, W. Schulte, and D. Noktin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Proc. the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), April 2005.

[26] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. Technical Report UW-CSE-04-10-03, University of Washington Department of Computer Science and Engineering, Seattle,WA, Oct. 2004.

[27] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspect-oriented programs. Technical Report NDSU-CS-TR04-XU03, North Dakota State University Computer Science Department, September 2004.

[28] J. Zhao. Tool support for unit testing of aspect-oriented software. In Proc. OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development, Nov. 2002.

[29] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In Proc. 27th IEEE International Computer Software and Applications Conference, pages 188–197, Nov. 2003.

[30] Y. Zhou, D. Richardson, and H. Ziv. Towards a practical approach to test aspect-oriented software. In Proc. 2004 Workshop on Testing Component-based Systems (TECOS 2004), Net.ObjectiveDays, Sept. 2004.

[31] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. ACM Comput. Surv., 29(4):366–427, 1997.

[32] M. Weiser. Program slicing. In International Conference on Software Engineering Proceedings, pages 439 449, 1981.

[33] P. McMinn. Search-based software test data generation: A survey. Software Testing, Verification and Reliability, 14(2):105--156, 2004.

Anuranjan Misra is working as Professor and Dean Academics at Bhagwant Institute of Technology, Ghaziabad, India. He had authored 20 books on computer engineering and its application areas. He had been reviewed many research papers conferences organized by IEEE computer society, Los Angles. He has been published many research papers in reputed Journals of India and aboard. He had member of many international professional societies.

Raghav Mehra is working as Associate Professor, at Bhagwant Institute of Technology, Muzaffarnagar, India. He has been published many research papers in reputed Journals of India and aboard. He had member of many international professional societies.

Mayank Singh is working as Assistant Professor, Computer Science Department, J.P. Institute of Engineering & Technology, Meerut, India.

Jugnesh Kumar is working as Assistant Professor at Manav Rachna College of Engineering, Faridabad, India.

Shailendra Mishra is working as Professor and Head of Department of Computer science and Engineering at kec dwarahat uttarakhand