An Optimized Approach of Fault Distribution for Debugging in Parallel

Maneesha Srivasatav*, Yogesh Singh** and Durg Singh Chauhan***

Abstract—Software Debugging is the most time consuming and costly process in the software development process. Many techniques have been proposed to isolate different faults in a program thereby creating separate sets of failing program statements. Debugging in parallel is a technique which proposes distribution of a single faulty program segment into many fault focused program slices to be debugged simultaneously by multiple debuggers. In this paper we propose a new technique called Faulty Slice Distribution (FSD) to make parallel debugging more efficient by measuring the time and labor associated with a slice. Using this measure we then distribute these faulty slices evenly among debuggers. For this we propose an algorithm that estimates an optimized group of faulty slices using as a parameter the priority assigned to each slice as computed by value of their complexity. This helps in the efficient merging of two or more slices for distribution among debuggers so that debugging can be performed in parallel. To validate the effectiveness of this proposed technique we explain the process using example.

Keywords—Clustering, Debugging, Fault Localization, Optimization, Software Testing

1. INTRODUCTION

Debugging is the most expensive, time consuming and dominantly manual process for software developers. The cost related to debugging is measured mostly on two parameters: (a) manual labor and (b) time required to discover and correct bugs to produce a failure free program. The primary reason for the high cost of debugging is the manual effort required to localize and remove faults and the time consumed in producing failure free software. For an efficient debugging process developers always try to find a good trade-off between the two. Among all debugging activities, fault localization is among the most expensive [1].

When software fails it is usually due to more than one cause. At the time of failure debuggers are not aware of the number of causes one failure might have. Thus, usually a one-bug-at-a-time debugging approach is carried out in a sequential manner to locate a fault and then to fix it. In this approach the debugger might utilize data from failed test cases and apply a fault localization technique where one bug is targeted at a time. After localizing and fixing the fault the program is retested, which might lead to another failure causing the cycle to be repeated until the program becomes failure free.

Manuscript received August 20, 2010; first revision November 4, 2010; accepted November 18, 2010.
Corresponding Author: Maneesha Srivastav
* Dept. of Computer Science and Engineering/Information Technology, Jaypee Institute of Information Technology, Noida, India (ek.maneesha@gmail.com)
** University School of Information Technology, Guru Gobind Singh Indraprastha, University, Kashmere gate, Delhi, India (ys66@rediffmail.com)
*** Uttarakhand Technical University, Dehradun, India (pdschauhan@gmail.com)

However, when there are multiple debuggers available for the debugging task we can create more specialized test suites based on fault focusing clusters and then distribute debugging tasks. By distributing debugging tasks we can save time and hence make the debugging activity less expensive. [2] Have presented a new mode of debugging technique that provides a way for multiple developers to simultaneously debug a program of multiple faults by automatically producing specialized test suites that target individual faults. This technique has been termed 'parallel debugging'. Debugging in parallel reduces the time required to debug multiple faults in a program. It involves distributing the program into many executable faulty slices which can be debugged independently. Multiple debuggers are then allocated these individual slices for independent debugging.

In this research we propose a new debugging technique called faulty slice distribution which will help in minimizing the cost related to debugging as measured by two parameters, namely (a) manual labor - as we are seeking to minimizing costs related to debugging by necessitating the minimum amount of debuggers and (b) the time required to discover and correct bugs - as our algorithm will select code based on the complexity estimation approach presented in [3] to efficiently distribute tasks among debuggers. This way debugging can be performed in parallel (i.e. simultaneously) with respect to other debuggers. The main contribution of this work is:

- A method to group related faulty slices for minimum redundancy in the debugging process for parallel debugging.
- A method to calculate the required number of debuggers for the minimum time spent in the debugging process.
- A method to create the required number of groups of faulty slices when the number of debuggers is limited.

2. RELATED WORK

Much of the recent work in debugging has been focused on fault localization as it is one of most expensive parts of the debugging practice. There are various coverage-based fault localization techniques aiming at identifying executing program elements. Among them some use coverage information provided by test suites to locate faults. Such techniques [4-7] typically implement and execute programs with test suites in order to gather runtime information. Other faults localization techniques are: χSlice [8] which collects coverage from failed test runs and passed test runs and then uses the set of statements executed only from the failed test run to be reported as likely faulty statements, Nearest Neighborhood (NN) [9] is an extension of [8] which features an extra step of passed test run selection. Tarantula [10] defines a color scheme to measure correlations i.e. it searches for those statements whose coverage has a relatively strong (but not necessarily strict) correlation with program failures. The empirical comparison of [11] compares Tarantula with χSlice and NN and their results show that Tarantula performs best among them. Statistical debugging [12, 13] implements predicates in the program and locates faults by comparing evaluation results from the predicates in failed test runs with those in all test runs whereas Delta debugging [14, 15] grafts values from a failed test run to a passed test run. However the cost of repeating trials can be expensive [16] but it has been shown to be useful in revealing many real world faults. The results [11] show that when multiple test runs are avail-

able, the performance of Coverage Based Fault Localization (CBFL) is better than that of delta debugging.

3. BASIC TERMINOLOGY AND OUR PROPOSED APPROACH

Locating a fault and debugging it not an easy task. It is costly as well as time consuming. The primary reason for the high cost of debugging is the manual effort required to localize and remove faults and the time consumed in producing failure free software. We here present an approach that will help in minimizing the cost related to debugging by allowing concurrent debugging of each faulty slice. This will in turn minimize the time required to discover and correct bugs, thus achieving an improvement in the two most troublesome aspects of debugging - cost and time. The whole process of work distribution is divided into a 4-stage distribution hierarchy as shown in Fig 1.

The process starts with the Fault localization stage. In order to distribute tasks efficiently, we use [10] a fault localization technique to generate different faulty slices focused around individual faults. The second stage considers the results of stage 1 and combines these faulty slices to generate different clusters that are to be debugged in parallel using [2]. The third stage uses the clusters generated in stage 2 and computes an estimation of the complexity of each slice using [3]. This estimation will be helpful in distributing the debugging tasks among debuggers. We propose this method for efficiently distributing debugging tasks such that every debugger gets approximately an equal amount of work; this is done in stage 4. A brief overview of each stage is presented below:

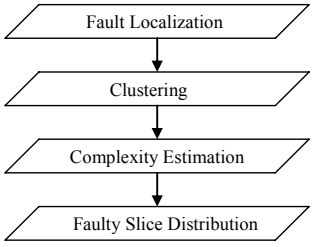

Fig. 1. 4-Stage Distribution Hierarchy

3.1 Stage 1: Fault Localization

Developers all over the world have been using a number of fault localization techniques for effective and efficient fault localization. One such technique is the coverage based fault localization technique (CBFL).It usually executes program (after its implementation) using a test suite to gather information at runtime. With every test case the implemented program records (a) program entities which were executed and (b) whether test cases passed or failed. Based on this information suspiciousness of every entity is generated which is then used to sort the entities into a decreasing order of suspiciousness. This sorted list is then used to generate ranks of all entities, which can further be used by developers as guidance to find faulty code. For this research we use [17] to localize faults using the following formula:

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{tot\ failed\ x\ (\ failed(s) + passed(s))}} \quad (1)$$

This technique utilizes TARANTULA to calculate suspiciousness of a statement. This suspiciousness is calculated using the given formula where *failed(s)* is a function which returns the number of test cases which executed this statement and failed and similarly *passed(s)* indicates the number of test cases which passed. The intuition behind this approach is that a statement which manifests more in failing cases is more likely to be faulty.

A number of metrics have been proposed and utilized but we chose this one as it is indicated in [17] that the similarity coefficient Ochaia used in microbiology was found to be most efficient in [18]. In this formula, the total number of failing test cases is indicated by *totfailed*, failing test cases which cover statement s are called *failed(s)*, and passing test cases covering s are called *passed(s)*.

3.2 Stage 2: Clustering Technique

To group faulty slices we use clustering technique [2] which uses the Jaccard similarity metric to compute pairwise similarities among rankings generated in stage 1. It then clusters pairs which are marked as similar by taking their closure. The Jaccard similarity metric uses the following formula to compute *similarity* between two sets, *A* and *B*, such that:

$$Similarity = \frac{A \bigcap B}{A \bigcup B} \quad (2)$$

To calculate similarity between two slices we first find the most important faulty statements to be compared, this requires a threshold value

This clustering technique helps to combine similar faulty slices into one slice to minimize redundancy in debugging. Once this similarity is calculated we can decide a threshold value e.g. 0.6 or 0.8 for determining similarity, all the slices which are similar by more than the threshold are then combined in one cluster. Consider the examples given in Appendix 1 and 2; we have given the clusters generated by combining individual slices. For the program in Fig 5(a) the clusters generated are in 5(b),(c) and (d) and for the program in Fig. 6(a) the clusters generated are in Fig 6(b),(c),(d) and (e).

3.3 Stage 3: Complexity Estimation

In our previous work [3] we showed a technique to compute the complexity of a faulty slice. In that work the complexity of a faulty slice P was calculated by using suspiciousness of statements *(S_i)* using the following formula

$$Complexity(P) = \sum_{i=1}^{n} Suspicousness(s_i) \quad (3)$$

For an example, the programs given in Appendix 1 and 2 corresponding complexities are shown in the last row of each table.

3.4 Stage 4: Faulty Slice Distribution

In this stage our algorithm will distribute slices obtained from stage 3 for equal distribution among debuggers. It works on the principle of first selecting the slice of highest complexity so that the volume of each queue (bucket) can be estimated. This will help in distributing an equal amount of work among debuggers for parallel debugging.

Faulty slice distribution is a technique which enables multiple debuggers to debug a single program simultaneously as well as independently. This is accomplished by creating different faulty slices of a program and distributing these slices among debuggers. Since the individual faulty slices will be debugged simultaneously by these debuggers their work load should be comparable for the maximum efficiency and utilization of the debuggers' time and effort. So before distributing these faulty slices we will calculate the complexity of each faulty slice and then distribute these slices equally among the debuggers.

The whole process of estimating complexity, assignment of volume to each queue, estimation of the number of debuggers required and distribution of tasks is explained in section 5. Next we explain the framework adopted for our approach followed by the data collection performed by our faulty slice distribution algorithm to distribute tasks among debuggers. The experimental study and result analysis is presented in section 6 followed by an application of the presented approach and then the conclusion.

4. PROCESS MODEL

The framework for our proposed approach is presented in Fig. 2 and a description of its various components is given below.

(a) **Test case generator:** This module generates test suites for exhaustive testing such that every path of the program is tested. It takes a program as an input and generates a list of failed and successful test cases as output.

(b) **Program Slicer:** It reads the output of the test case generator and separates faulty test cases from correct ones. These faulty test cases are then stored as individual slices.

(c) **Cluster Slices:** It reads faulty slices from the program slicer and compares each slice to others and generates a matrix for estimating similarities among them. Based on this data it produces an optimized set of slices by combining all those slices whose statements overlap with other slices by more than the given threshold so that redundancy can be minimized.

(d) **Complexity Evaluator**: It calculates the complexity of each slice by calculating the suspiciousness of statements using our approach as presented in [3].

(e) **Optimum Debuggers**: In this phase slices are distributed among debuggers by analyzing the complexity estimations generated from the previous step. Since these slices are from the same program there are bound to be some dependence and similarities between them. Thus it will be helpful in estimating the total time required to debug a particular slice. This information can be used to calculate the minimum time required to debug an entire set in parallel which in turn can be used to calculate the minimum number of debuggers required to do the job. Hence, depending on criterions highlighted by the user (i.e., Number of debuggers, Amount of time) we can generate a chart to estimate the overall time

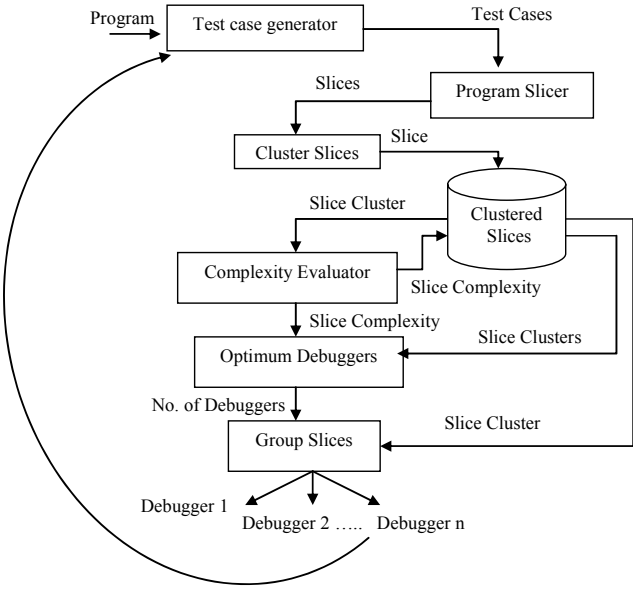

Fig. 2. Process Model

required for debugging and the number of debuggers needed.

(f) **Group Slices**: Depending upon the time calculated from the previous step for processing a slice(s) we can further combine two or more slices if the time required to process a slice is not distributed equally. In that case we will club two or more slices where the length of a slice will be the time required to process the longest slices in order to utilize time optimally.

5. FAULTY SLICE DISTRIBUTION

To distribute clustered faulty slices our algorithm first stores the estimated complexities of these slices and their respective priorities in ascending order in a database. It then analyzes on a priority basis the complexity of each faulty slice and selects the one with highest complexity, M_C from the table maintained in database similar to Table 1. This slice with M_C complexity is then taken and stored in a queue. This will help in defining a maximum limit or volume of queue M, where a queue with a capacity M means that it can hold faulty slices whose complexity sum does not exceed M. For subsequent queues we analyze Table 1 in the complexity base for the next higher priority such that:

$$(H_a + H_b) <= M_c \tag{4}$$

Where H_a is complexity of the next slice with second highest priority (highest M_C) and H_b is the one with lowest complexity in the database. We propose the following rules to fill up queues:

Table 1. View of Complexity base

Priority	Complexity
P_1	H_1
P_2	H_2
P_3	H_3
…	…
…	…
P_n	H_n

- The slices are arranged in decreasing order of their complexity to enable fast and efficient selection.
- If the sum of $(H_a + H_b)$ is less than M_C then these two slices will be stored in the next queue.
- If there is more than one slice which satisfies the above criteria then we select the slice which has maximum similarity with H_a.

The selection is done in such a manner that clusters added to a queue do not exceed its limit. This will help in optimizing the number of queues thereby minimizing the number of debuggers. This process of creating queues will be repeated until there are no more clusters to be allocated. The total number of queues thus formed is in fact the number of debuggers required to debug the program in minimum time.

5.1 Tool Proposed for Faulty Slice Distribution

We considered the issue of providing tool support for our proposed approach. The nature of this support is sketched in Fig. 3. As illustrated the database consists of two parts, the clustered slice base and the complexity slice base. These two bases are interlinked. Thus, it is possible to move to the complexity base where the complexity and priority assigned to each clustered faulty

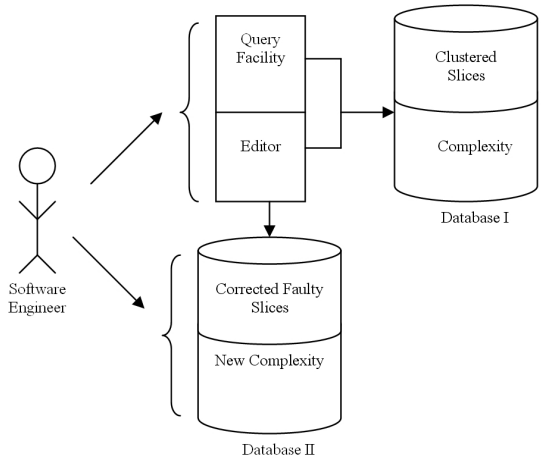

Fig. 3. Faulty Slice Distribution Tool Sketch

slice is stored and also to the clustered base for analyzing the actual slice for debugging. The information is stored in the form of a table as shown below (Table 1).

The Query Facility is to be used by a software engineer who can be a debugger or a leader of a team in order to navigate through these two bases for distribution of tasks among debuggers. The results of query are displayed in graphical form. This form is edited to locate a fault and debug it. For updating slices and their respective complexities we use another database to store this information. The reason for using another database is to store historic information of systems for future use (re-use). The two bases namely, the corrected faulty slices and new complexities are then populated with information about debugged slices. The software engineer can at any time access this information for analysis.

5.2 Algorithm

We propose the following algorithm to distribute faulty slices among debuggers. Firstly this algorithm finds maximum complexity among the set of faulty slices given (in steps 1 to 6). Maximum complexity thus attained is taken to be the approximate capacity of a queue of debugging tasks to be given to a debugger. Then it creates a queue of all faulty slices to be allocated to debuggers (steps 7 and 8).

The next step is to allocate these faulty slices. We take one faulty slice at a time and allocate it to a queue depending on its similarity to the contents of that queue and the space available in the queue (step 11 to 27). When there are no more faulty slices to be allocated to the queue we create one more queue (step 28 and 29) and start filling it with the remaining faulty slices. Finally this allocation stops once all the faulty slices are allocated.

The algorithm presented in Fig. 4 takes as an input the complexities of faulty slices generated

```
1.    F_i is i^th faulty slice
2.    C_i is complexity of i^th faulty slice
3.    max = C_1
4.    for i from 1 to n
5.            if (C_i > max)
6..                   max = C_i
7.    for i from 1 to n
8.            enqueue(S, F_i)
9.    i = 1
10.   while empty(S) = FALSE
11.           while (Complexity(Q_i)<max )
12.                   if (Q_i is empty)
13.                           enqueue(Q_i, dequeue(S))
14.                   else
15.                           C = start(S)
16.                           P = endof(Q_i)
17.                           sim = 0
18.                   while(( C != NULL) AND (Complexity(Q_i)+ Complexity(C))<max)
19.                           if(similarity(P,C) >sim)
20.                                   Z = C
21.                                   sim = similarity(P,C)
22.                                   C = C→next
23.                   if ( Z != NULL )
24.                           remove(S,Z)
25.                           add(Q_i, Z)
26.                   else
27.                           break
28.           i = i +1
29.           initialize(Q_i)
```

Fig. 4. Algorithm to distribute debugging tasks among debuggers to minimize time

as a result of complexity estimation. Then by utilizing the complexity estimation of each faulty slice it finally divides the set of faulty slices among debuggers. This algorithm is used in the fourth stage of the hierarchy defined in section 3. For the first, second and third stages of this method we have referred to the techniques used in [2, 3] and [17] as mentioned in the respective sections.

The execution time of the above algorithm depends on the number of faulty slices created and the maximum complexity of those faulty slices. For an average case one faulty slice will be visited a maximum of one time and hence if there are (n) faulty slices then this algorithm would take approximately O(n) order time. However, in cases where many slices have similar complexities this algorithm might take O (n^2) time, but such a case would be unique and very unlikely.

6. EXAMPLE AND DATA OBSERVATION FOR RESULT ANALYSIS

This section describes empirical data collected for analysis using the proposed strategy. The data is collected for the programs given in Appendix 1 and 2. From these codes respectively three and four clusters of faulty slices were generated. The details regarding the line of code and complexity of each slice is given in Table 2 below.

In [2] M. J. Harrold et.al have introduced the idea of debugging in parallel. They have elaborated on why debugging in parallel would be a better approach than sequential debugging. In our previous work [3] we had defined a method to estimate the amount of debugging work related to a given slice.

However, in all the papers listed in the literature no one addresses the problem of *equally* distributing debugging tasks among debuggers. Our approach defines a four-stage hierarchy that enables us to divide one program into several sets of faulty slices, then to estimate the complexity of each slice thereby estimating the amount of work to be done on each slice and then finally to distribute these slices among debuggers equally. This will minimize time required to debug a particular program, also each debugger gets a simpler task which is more focused on a single

Table 2. Complexity estimation of each slice in two examples

	Example 1			Example 2			
	Slice 1	Slice 2	Slice 3	Slice 1	Slice 2	Slice 3	Slice 4
Line of code	8	13	5	6	27	8	21
Complexity	2.92	5.56	1.45	4.98	17.9	4.81	15
Sum of complexities	9.93			42.69			
Highest Complexity (MC)	5.56			17.9			

Table 3. Efficient data distribution

	Example 1		Example 2		
	Debugger 1	Debugger 2	Debugger 1	Debugger 2	Debugger 3
	Slice 1	Slice 2	Slice 1	Slice 2	Slice 4
	Slice 3	--	Slice 3		
Complexity	4.37	5.56	9.82	17.9	15

fault hence it simplifies the debugging process for a debugger. The above example successfully presents the advantage of task distribution, unlike other existing debugging techniques.

7. CONCLUSION

In this paper we have presented an efficient approach for distributing tasks among debuggers. The entire process is divided into three stages. It first extracts faulty slices which are fault focused using [10] and then clusters these slices using a Jaccard similarity matrix [2]. In the next step it measures the complexity of each clustered slice to estimate time and labor required to debug them. This is done using our complexity estimation technique of [3]. Once the complexity of each slice is calculated certain priority is given to these slices having been calculated using the values of their complexity. Based on this data slices are grouped together with the maximum capacity of a group being equal to the highest complexity of a slice.

REFERENCE

[1] I. Vessey, "Expertise in Debugging Computer Programs," International Journal of Man-Machine Studies: A process analysis, Vol.23(5), 1985, pp.459-494.

[2] James A. Jones, James F. Bowring and Mary Jean Harrold, "Debugging in Parallel," Proc. ACM. International Symposium on Software Testing and Analysis (ISSTA 07), July, 2007.

[3] M. Srivastav, Y. Singh, C. Gupta, D.S. Chauhan, "Complexity Estimation Approach for Debugging in Parallel", Proceedings of IEEE - 2010 Second International Conference on Computer Research and Development, Kuala Lumpur, Malaysia, May, 2010.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-Based Fault Localization," Proc. Testing: Academic & Industrial Conference Practice And Research Techniques (TAIC PART-MUTATION 07), IEEE Computer Society, September, 2007, pp.89-98.

[5] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 05), November, 2005.

[6] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," Proc. ACM International Conference on Software Engineering (ASE 02), May, 2002.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," Proc. ACM SIGPLAN. Programming Language Design and Implementation (PLDI 05), June, 2005.

[8] H. Agrawal, J. Horgan, S., Lodon, and W. Wong, "Fault Localization using Execution Slices and Dataflow Tests," Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE 95), October, 1995.

[9] M. Renieris, and S. Reiss, "Fault Localization with Nearest Neighbor Queries," Proc. IEEE International Conference on Software Engineering (ASE 03), October, 2003.

[10] J.A. Jones, M.J. Harrold, and J. Stasko, "Fault Localization using Visualization of Test Information," Proc. International Conference on Software Engineering (ICSE 04), IEEE Computer Society. May, 2004.

[11] J.A. Jones, and M.J. Harrold, "Empircal Evaluation of the Tarantula Automatic Fault-Localization Technique," Proc. ACM International Conference on Software Engineering (ASE 05), November, 2005.

[12] B. Liblit, A. Aiken, A.X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," Proc. ACM SIGPLAN. Conference on Programming Language Design and Implementation (PLDI 03), June, 2003, pp.141-154.

[13] C. Liu, L. Fei, X.F. Yan, J.W. Han, and S. Midkiff, "Statistical Debugging: a Hypothesis Testing-Based Approach," IEEE Transactions on Software Engineering, Vol.32(10), 2006, pp.1-17.
[14] H. Cleve, and A. Zeller, "Locating Causes of Program Failures," Proc. International Conference on Software Engineering (ICSE 05), IEEE Computer Society. May, 2005.
[15] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," Proc. ACM SIGSOFT. Fast Software Encryption (FSE 02), November, 2002, pp.1-10.
[16] X.Y. Zhang, S. Tallam, N. Gupta and R. Gupta, "Towards Locating Execution Omission Errors," Proc. ACM SIGPLAN. Programming Language Design and Implementation (PLDI 07), June, 2007, pp.415-424.
[17] R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold, "Lightweight Fault-Localization Using Multiple Coverage Types" Proc. International Conference on Software Engineering (ICSE 09), IEEE Computer Society. May, 2009.
[18] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. "On the accuracy of spectrum-based fault localization" Proc. Of TAIC-PART '07, September, 2007, pp.89-98.

Maneesha Srivastav

She is a Senior lecturer at Jaypee Institute of Information Technology, India. She holds a Masters of Technology and Bachelor's of Information Science degrees in Computer Science and Engineering. Her areas of interest are Software Engineering, Software Debugging, Software Project Management, Data Structures and Algorithms. She has published papers in international journals and conferences as well. Currently she is pursuing her Ph.D. in Software Debugging.

Yogesh Singh

He received his master's degree and Ph.D. degree in Computer Engineering from the National Institute of Technology, Kurukeshtra, India. He is a professor at University School of Information Technology (USIT), Guru Gobind Singh Indraprastha University, Delhi, India. His research interests include software engineering focusing on the area of Software project planning, Testing, Metrics, Data Structures, Computer Architecture, Parallel Processing and Neural Networks. He is also a Controller of Examinations with the Guru Gobind Singh Indraprastha University. He was founder Head and dean of the University School of Information Technology of Guru Gobind Singh Indraprastha University. He is co-author of a book on software engineering, and is a Fellow of IETE and member of IEEE. He has more than 200 publications in international and national journals and conferences.

Durg Singh Chauhan

He received his Ph.D. degree from Indian Institute of Technology (IIT) Delhi in 1986, India and did his post doctoral work at Goddard space Flight Centre, Greenbelf Maryland. USA (1988-91). He is a Vice-Chancellor of Uttarakhand Technical University, Dehradun, India. Prior to this he had also served as Vice-Chancellor in three other universities in India. He has been a member of the University Grant Commission (UGC), National Board of Accreditation (NBA) – executive, All India Council for Technical Education (AICTE), Council for Advancement of People's Action and Rural Technology (CAPART), National Accreditation Board for Testing and Calibration Laboratories (NABL) - Department of Science and Technology (DST) executive and member, National expert Committee for IIT- (National Institute of Technology) NIT research grants. He has authored two books and published and presented more than 115 research papers in international journals and international conferences and wrote more than 20 articles on various topics in national magazines. He has delivered hundreds of lectures in U.S. and Canadian universities.

APPENDIX 1, EXAMPLE 1

Test Cases		t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	
Values of Variables	ch	1	2	2	3	6	1	1	2	2	2	3	
	n	-1	3	6	3	6	2	0	5	4	7	4	
switch(ch){		1	1	1	1	1	1	1	1	1	1	1	0.71
case 1 : printf("\n\t Enter Number for Factorial : ");		1	0	0	0	0	1	1	0	0	0	0	0.41
scanf("%d", &n);		1	0	0	0	0	1	1	0	0	0	0	0.41
i=1;		1	0	0	0	0	1	1	0	0	0	0	0.41
while(n>=0){ // n>0		1	0	0	0	0	1	1	0	0	0	0	0.41
i*=n--;}		0	0	0	0	0	1	1	0	0	0	0	0.42
printf("\n\t The Factorial is : %d", i);		1	0	0	0	0	1	1	0	0	0	0	0.41
break;		1	0	0	0	0	1	1	0	0	0	0	0.41
case 2 : printf("\n\t Enter Number to check : ");		0	1	1	0	0	0	0	1	1	1	0	0.50
scanf("%d", &n);		0	1	1	0	0	0	0	1	1	1	0	0.50
i=2;		0	1	1	0	0	0	0	1	1	1	0	0.50
while(i<n){		0	1	1	0	0	0	0	1	1	1	0	0.50
if(n%i == 2){ // n %i ==0		0	1	1	0	0	0	0	1	1	1	0	0.50
printf("\n\t Number is not prime ");		0	1	1	0	0	0	0	1	0	1	0	0.40
break;		0	1	1	0	0	0	0	1	0	1	0	0.40
}		0	1	1	0	0	0	0	1	0	1	0	0.40
i++;		0	1	1	0	0	0	0	1	1	1	0	0.50
}		0	1	1	0	0	0	0	0	1	0	0	0.27
if(i == n)		0	1	1	0	0	0	0	1	1	1	0	0.50
printf("\n\t Number is Prime");		0	1	0	0	0	0	0	0	1	0	0	0.28
break;		0	1	1	0	0	0	0	1	1	1	0	0.50
case 3 : printf("\n\t Enter the Number to check : ");		0	0	0	1	0	0	0	0	0	0	1	0.28
scanf("%d", &n);		0	0	0	1	0	0	0	0	0	0	1	0.28
if(n%2 == 0)		0	0	0	1	0	0	0	0	0	0	1	0.28
printf("\n\t Number is Odd");// Number is Even		0	0	0	0	0	0	0	0	0	0	1	0.30
else		0	0	0	1	0	0	0	0	0	0	0	0
printf("\n\t Number is Odd ");		0	0	0	1	0	0	0	0	0	0	0	0
break;		0	0	0	1	0	0	0	0	0	0	1	0.28
default : printf("\n\t Error in input");		0	0	0	0	1	0	0	0	0	0	0	0
}//End of Switch		1	1	1	1	1	1	1	1	1	1	1	0.71
Success (s) of Failure(f)		s	s	s	s	s	f	f	f	f	f	f	

cluster 1 cluster 2 cluster 3

Fig. 5. (a) Program to perform three calculating tasks

case 2 : printf("\n\t Enter the Number for checking Prime : ");	0.50
scanf("%d", &n);	0.50
i=2;	0.50
while(i<n){	0.50
if(n%i == 2){ // n %i ==0	0.50
printf("\n\t Number is not prime ");	0.40
break;	0.40
}	0.40
i++;	0.50
}	0.27
if(i == n)	0.50
printf("\n\t Number is Prime");	0.28
break;	0.50
Complexity	**5.56**

Fig. 5. (b) Slice 1

case 3 : printf("\n\t Enter the Number to check : ");	0.28
scanf("%d", &n);	0.28
if(n%2 == 0)	0.28
printf("\n\t Number is Odd"); //Number is Even	0.30
break;	0.28
Complexity	**1.45**

Fig. 5. (c) Slice 2

APPENDIX 2, EXAMPLE 2

Values of Variables	first	K	B	B	J	J	B	A	D	B	I	
	second	B	K	E	A	C	J	C	E	C	E	
if(first-65 > 8 || second - 65 > 9);		1	1	1	1	1	1	1	1	1	1	0.84
else if(web[first-65][second-67])		0	0	1	0	0	1	1	1	1	1	0.79
{		0	0	0	0	0	0	0	1	0	0	0.38
printf("\nDirect connection from %c to %c ",first,second);		0	0	0	0	0	0	0	1	0	0	0.38
con=1;		0	0	0	0	0	0	0	1	0	0	0.38
}		0	0	0	0	0	0	0	1	0	0	0.38
else		0	0	1	0	0	1	1	0	1	1	0.71
{		0	0	1	0	0	1	1	0	1	1	0.71
for(i=0;i<10;i++)		0	0	1	0	0	1	1	0	1	1	0.71
{		0	0	1	0	0	1	1	0	1	1	0.71
if (web[first-65][i])		0	0	1	0	0	1	1	0	1	1	0.71
{		0	0	1	0	0	1	1	0	1	1	0.71
if(web[i][second-65]&&con!=1)		0	0	1	0	0	1	1	0	1	1	0.71
{		0	0	0	0	0	0	0	0	1	1	0.53
printf("\nConnection through %c->%c->%c", first, i+64, second);		0	0	0	0	0	0	0	0	1	1	0.53
con=0;		0	0	0	0	0	0	0	0	1	1	0.53
}		0	0	0	0	0	0	0	0	1	1	0.53
else		0	0	1	0	0	1	1	0	0	0	0.50
{		0	0	1	0	0	1	1	0	0	0	0.50
if(con!=1)		0	0	1	0	0	1	1	0	0	0	0.50
{		0	0	1	0	0	1	1	0	0	0	0.50
for(j=0;j<9;j++)		0	0	0	0	0	1	1	0	0	0	0.53
{		0	0	1	0	0	1	1	0	0	0	0.50
if((i!=j)&&(i!=first-65)&&(web[i][j]&&web[j][second-65])&&con!=1)		0	0	0	0	0	1	1	0	0	0	0.53
{		0	0	1	0	0	0	0	0	0	0	0.00
printf("\nConnection through %c->%c->%c->%c",first,i+65,j+65,second);		0	0	1	0	0	0	0	0	0	0	0.00
con=1;		0	0	1	0	0	0	0	0	0	0	0.00
}		0	0	1	0	0	1	1	0	0	0	0.50
}		0	0	1	0	0	1	1	0	0	0	0.50
}		0	0	1	0	0	1	1	0	0	0	0.50
}		0	0	1	0	0	1	1	0	1	1	0.71
}		0	0	1	0	0	1	1	0	1	1	0.71
}		0	0	1	0	0	1	1	0	1	1	0.71
}		1	1	1	1	1	1	1	1	1	1	0.84
if(con==0)		1	1	1	1	1	1	1	1	1	1	0.84
{		1	1	0	1	1	1	1	0	1	1	0.82
printf("\nBAD BAD PAGE! NO DOUGHNUT FOR YOU!!");		1	1	0	1	1	1	1	0	1	1	0.82
}		1	1	0	1	1	1	1	0	1	1	0.82
Result : Success (s) and Failure(f)		s	s	s	f	f	f	f	f	f	f	

Fig. 6. (a) Program to find a path from one point to another, indicates four clusters are formed

if(first-65 > 8 || second - 65 > 9);	0.84
}	0.84
if(con==0)	0.84
{	0.82
printf("\nBAD BAD PAGE! NO DOUGHNUT FOR YOU!!");	0.82
}	0.82
Complexity	**4.98**

Fig. 6. (b) Slice 1

if(first-65 > 8 || second - 65 > 9);	0.84
else if(web[first-65][second-67])	0.79
{	0.38
printf("\nDirect connection from %c to %c ",first,second);	0.38
con=1;	0.38
}	0.38
}	0.84
if(con==0)	0.84
Complexity	**4.81**

Fig. 6. (c) Slice 3

if(first-65 > 8 || second - 65 > 9);	0.84
else if(web[first-65][second-67])	0.79
else	0.71
{	0.71
for(i=0;i<10;i++)	0.71
{	0.71
if (web[first-65][i])	0.71
{	0.71
if(web[i][second-65]&&con!=1)	0.71
{	0.53
printf("\nConnection through %c->%c->%c",first,i+64,second);	0.53
con=0;	0.53
}	0.53
}	0.71
}	0.71
}	0.71
}	0.84
if(con==0)	0.84
{	0.82
printf("\nBAD BAD PAGE! NO DOUGHNUT FOR YOU!!");	0.82
}	0.82
Complexity	**15**

Fig. 6. (d) Slice 2

if(first-65 > 8 || second - 65 > 9);	0.8
else if(web[first-65][second-67])	0.8
else	0.7
{	0.7
for(i=0;i<10;i++)	0.7
{	0.7
if (web[first-65][i])	0.7
{	0.7
if(web[i][second-65]&&con!=1)	0.7
else	0.5
{	0.5
if(con!=1)	0.5
{	0.5
for(j=0;j<9;j++)	0.5
{	0.5
if((i!=j)&&(i!=first-65)&&(web[i][j]&&web[j][second-65])&& con!=1)	0.5
}	0.5
}	0.5
}	0.5
}	0.7
}	0.7
}	0.7
}	0.8
if(con==0)	0.8
{	0.8
printf("\nBAD BAD PAGE! NO DOUGHNUT FOR YOU!!");	0.8
}	0.8
Complexity	**17.9**

Fig. 6. (e) Slice 4