Test-Case Minimization Technique for Large Scale Software using Link-Matrix

Umesh Kumar Tiwari, Santosh Kumar, Kamlesh Purohit, Vijay Kumar

Abstract*: *In today's software development environment, testing commences just after the finalization of systems requirements. Testing process is executed to make the underlying software error free, to enhance and assure the development of reliable and quality software product. This paper explores and analyses some of black-box as well as white-box testing methods according to their key findings, measures and metrics and considerations of testing factors. Further this paper proposes an efficient test case minimization technique. This technique uses Link-Matrix to identify the number of actual parameters used between components. We have used Boundary Value Analysis (BVA) method to compute the count of tests cases for individual modules or components. We have compared the proposed technique with all three cases of Boundary Value Analysis method including normal case, robust case and worst case. Results achieved during comparison shows that this proposed technique is effective to minimize the total count of test-cases especially in case of robust and worst scenarios.

Index Terms*: *Testing, Link-Matrix, Large-scale software, black-box, white-box, Boundary value analysis.

I. INTRODUCTION

"Testing is the process of executing a program with the intent of finding errors." [1]. Testing is an effort to construct the software under consideration without errors. Identifying and fixing errors in early phases of development is always helpful to minimize the overall cost of the development. Testing software is an expansive endeavour by means of cost, effort and time [2]. Testing is a critical and crucial phase of the overall development of software constructs. Testing is the fundamental activity to verify the correctness, precision and compatibility of the software not only at the basic smallest level but also at the system level. Practitioners identified that the improper testing results to untrustworthy and undependable products [3], [4].

Testing is commonly used to *verify* and *validate* software [5], [6], [1]. *Verifying* components in CBSE constitutes the collection of procedures to certify the functionalities of components at individual level. V*alidating* components include the group of procedures to assure the integrity of integrated components according to the architectural design and fulfilling the needs of the customer. Testing methods must incorporate planning of testing, test-case design, implementation of testing, testing results and the assessment of collected data. Typically software is made up of different parts or sub-components. And these subparts are not only independently deployed but are from various contexts. Therefore we need testing techniques which must address not only the subparts individually but the whole software.

Testing starts at the component level and move forwards to the integrated complex higher level. In this work we have provided a critical survey on testing methodologies including individual standalone programs as well as complex software. For the testing purpose, software constructs are divided into two broad categories: *Input/Output constructs* and *Process/Logic/Code constructs*. Input/Output constructs refer to inputs supplied to the module/software and outcomes produced by the module/software. Normally input/output is presented in the form of data, information and specific values. Process/Logic/Code constructs refer to the actual processing of software. It involves the coding and internal structure of the software. According to the types of software constructs, testing techniques are organized into two major classes: black-box and white-box testing.

A. Black-Box Testing

Black-box testing techniques consider only inputs and outputs of the software. These techniques are applicable to that software whose code is not available or accessible. They divide input and output domains in various classes or partitions and test them separately. Black-box put emphasis on the external behaviour of the module/software. Figure 1 shows the overview of black-box testing. Testers control is on input-parameters and output-parameters only. The internal logic of the program and software are not accessible.

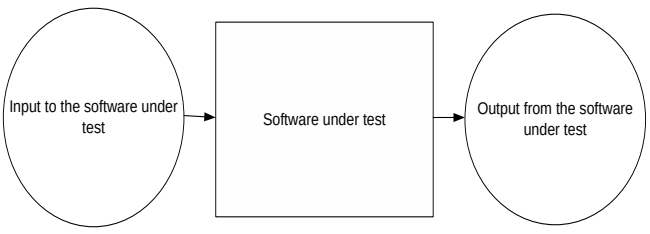

Fig. 1. Black-box Testing

Input to a program/software, which is under consideration, is given in the form of data, information or specific values as per the requirements.

Revised Manuscript Received on March 20, 2019.

Umesh Kumar Tiwari, Computer Science and Engineering, Graphic Era Deemed to be University, Dehradun, India.

Santosh Kumar, Computer Science and Engineering, Graphic Era Deemed to be University, Dehradun, India.

Kamlesh Chand Purohit, Computer Science and Engineering, Graphic Era Deemed to be University, Dehradun, India.

Vijay Kumar, Dpartment of Physics, Graphic Era Hill University, Dehradun

After processing the achieved outputs are collected and tested. Since the logic is hidden from the testers, their control is only on the inputs/outputs of the software.

B. White-Box Testing

White-box testing techniques consider the inner logic of the module/software. They are applicable to the structural code of the software. In this technique the program statements are checked and errors are fixed. White-box testing techniques are used to address the testing of the structural design and internal code of the software. Figure 2 shows the overview of white-box testing method. In white-box testing not only the structural behaviour of software is tested but the control flows, basis paths, data structures (within sub component and outside the sub components), independent paths, logical mistakes, coding errors, incoming and outgoing interfaces and semantic errors are also considered. Here input provided to the software is assumed to be true and the functionalities provided by the software are checked.

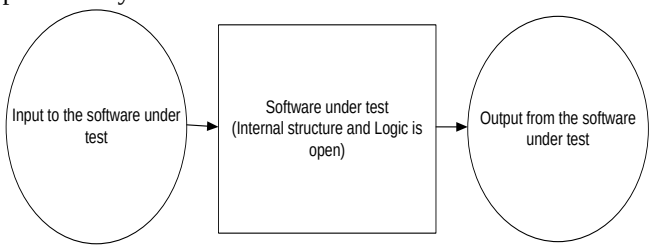

Fig. 2. Black-box Testing

Organization of the work is as follows; section I describes the introduction of the basic domain of the paper, section II covers the detailed literature review of black-box as well as white-box methods. Section III elaborates the proposed test case minimization technique including case study and comparative analysis with BVA methods. Finally section IV concludes the work.

II. LITERATURE SURVEY

We have performed the literature survey on the basis of program/software testing behaviour. Testing behaviours are divided into two classes: *black-box* and *white-box.*

A. Survey of Black-box Methods

Black-box testing methods focus on the behaviour of the software. In this technique the functional behaviour of the software is assessed through inputs provided and outputs observed. Black-box testing treats the internal logic of code as a black-box and the testing observations are captured using inputs and outputs only. Following are the some black-box testing strategies proposed in the literature [7, 8, 9, 10, 11, 12, 13] and classified as: i. Boundary-Value Analysis (BVA), ii. Equivalence-Class Partitioning (ECP). Iii. Decision Table-Based Testing (DTB), and iv. Cause-Effect Graphing (CEG).

i. Boundary-Value Analysis (BVA)

In the Boundary-Value Analysis (BVA) testing, assumption is that rather than within the limits of the conditions, errors appear on the external-boundaries. In BVA testing methodology emphasis is on verifying that the software behaves normally (accurately) at the extreme boundaries [7], [8]. Boundary value analysis method is used to draw test cases focusing on the input values at the "edges" of the logical conditions [9]. This method usually tests the centre values, boundary values, as well as the values just below the boundary and just above the boundary.

ii. Equivalence-Class Partitioning (ECP)

The Equivalence-Class Partitioning (ECP) is a black box testing strategy based on the assumption that if we divide the input values into classes or partitions on the basis of their validity or invalidity then the impact of each value under the particular class will be equivalent. That is the validity of every data member of the legitimate class is equal or similar; and every data member of an illegitimate class is equally invalid. In ECP the input domain is partitioned and tested for validity. Usually ECP is applied to the input classes, but in sometimes it is applied to the output classes also [10], [11].

iii. Decision Table-Based Testing (DTB)

The Decision Table-Based Testing (DTB) testing method is used to test the permutations of input conditions rather than testing single input values. Decision-table based testing was proposed to overcome the limitations of the BVA and the ECP. The limitation is that the BVA and ECP are applicable only when we have to test single values of input and output domain. The DTB testing method combines two or more than two input logical conditions and assesses and examines these complex conditions. There are four quadrants in Decision table, Condition stub: combine the input conditions, Action stub: specify the output conditions, Condition entries: input the entries of condition entries and Action entries: input the action entries, shown in Table 1.

Table 1. Decision-Table

Four quadrants	
Condition-Stub	Condition-entries
Action-Stub	Action-entries

In the DTB method, the Condition-stub and Condition-entries correspond to an input condition or set of input conditions. The Action-stub and Action-entries correspond to output or set of outputs [12], [13].

iv. Cause-Effect Graphing

The Cause effect-graphing method is also used to combine the input conditions so that more elaborative assessment can be performed. Here in this technique, inputs are recognized as causes and outputs are recognized as effects. We draw a Boolean graph known as cause-effect graph by joining these causes (inputs) and effects (effects). It is a simple graphing technique where nodes represent input conditions and edges represent linking between nodes.

Michael R. Lyu et al. [14] proposed that the system's effectiveness testing can be increased by regulating the factors of the components. These factors that eligible for optimization are "cost, reliability, effort, and similar attributes of the system components". Authors considered single as well as multiple application systems for "software component testing resource allocation". They used "reliability-growth curves" to model the association between the failure rates and "cost to decrease this rate".

Authors used interaction among components and and failure rates of components in their methodology.

Gordon Fraser and Andrea Arcuri [15] proposed an approach of covering all coverage goals of testing at a time rather than on coverage goal at a time. Authors proposed that covering all coverage goals at a time leads to a better and effective result in terms of minimizing test suites. For their work, authors have used object-oriented software.

Jehad Al Dallal and Paul Sorenson [16] introduced a method called "all-paths-state" to produce state-based test cases through testing at class level. They identified Framework Interface Classes to cover the maximum number of specifications. Authors suggested that this technique will help to generate reusable test cases effectively. They also proposed that this frame work design is equivalently helpful for reusable modules as well as newly developed modules.

Umesh K. and Santosh K. [17] proposed a black-box testing and test case generation method for component-based software. Their assumption is that when components interact with each other then interaction produces some effects in the form of correct outputs or errors. They offered method named Integration-effect graph. This method covers the input and output domains of the software.

B. Survey of White-box Methods

White-box techniques are used to address the testing requirements of the structural design and internal code of the software. White-box testing methods ensure that all execution paths and independent logics of the program or module have been tested adequately. This testing is also responsible for testing the logical decisions given in the code must be tested on their all sides, looping and branching codes at their extremes as well as within the boundaries must be checked at least once. For the analysis of the structure of the software, we have *Basis Path testing* technique, which is used in white-box technique. Thomas J. McCabe [18] proposed a formula to count the number test cases of a program. Author has given the Cyclomatic complexity based on the structural design of the code. He defined a control-flow graph having '*n' nodes*, '*e'* edges and *p* connected components. Cyclomatic complexity is calculated as $V(G) = e - n + 2p$, here 2 is the "result of adding an extra edge from the exit node to the entry node of each component module graph" [19].

Henderson-Sellers [20] proposed another formula to compute the Cyclomatic complexity. This proposed formula was an amendment to the McCabe's formula for complexity [19]. Henderson-Sellers' defined, "$V(G) = e - n + p + 1$". This altered formula argued that to make components strongly linked, an edge is added, and is denoted as constant 1 to the multi-component flow graph [21]. Author used the concept of modularity. A. Orso et al. [22] suggested a technique of using component's metadata in the testing of components. This metadata consists of data and control dependencies, source code, complexity metrics, security attributes, information retrieval mechanisms and execution procedures. Authors suggested using this metadata in regression testing of components and their interfaces. They also proposed a Specification-based Regression Test Selection technique for CBS. Bixin Li et al. [23] developed a "matrix-based" method to compute the dependencies in large scale software. Authors identify some categories of dependencies available in component-based software and they also defined "component dependence graph and the dependence matrix" to record the dependencies. On the basis of dependence-graph and dependence-matrix authors proposed a mathematical foundation for assessing these dependencies in component-based software. Gill and Tomar [24] presented their work focusing on testing requirements and documentation process of test cases of component-based software. Authors suggested that the access to the source code, functions provided by the component, compatibility with other components, components middleware, interactions made by various reusable components, specifications of the components, as the testing requirements of large-scale software. Authors also discussed the boundaries of the testing of large-scale software. Tamal Sen and Rajib Mall [25] proposed a testing technique based on state and dependencies of modules/software. This technique is called as "regression test selection technique" targeting the regression test suite. They argued that this method will reduce the number of test cases in regression testing. Authors' assumption is that the link between the state model in design phase and the executable code is very strong, and while maintaining the design state model code is altered without affecting the state of the component. Stephen H. Edwards [26] described test case methods for black-box and white-box components in CBSE. This technique lied on the concept of generating flow-graph from the specifications of components, and then applying conventional graph coverage methods. Authors focused on test case design using pre-condition, post-condition, and fault detection ratios. Herv'e Chang et al. [27] proposed a technique to built and install reusable integration connectors effectively. They define these connectors as the solution of problems arises during the integration of off-the-shelf components. Authors defined that these problems in the form of exceptions. These connectors are usually installed on the basis of integration information available with the components. They suggested the technique so that these connectors can heal exceptions automatically. They provide the complete structure, exception behaviour and the integration behaviour of these healing connectors.

Umesh T. and Santosh K. [21], proposed a Cyclomatic complexity computation method including the complexities of the standalone components and the component-based software. Authors used graph theory notations and interactions among components to calculate the Cyclomatic complexity of complex, multi-component software.

III. PROPOSED TEST-CASE MINIMIZATION TECHNIQUE

Proposed test case minimization technique is applicable to black box as well as white box software. This proposed technique is not for individual programs; rather it is used for the software which consists of various modules. To compute the count of test-cases for individual components we use Boundary Value Analysis (BVA) technique.

In case of normal Boundary Value Analysis (BVA) method, number of test cases for a program may be calculated as = 4 * n + 1, where, n denotes number of variables in the program.

In robust case of BVA, number of test cases = 5 * n + 1.

In worst case of BVA, number of test cases = 5^n.

These methods work well when we have to compute test cases for individual programs or components. But if we have to compute number of test cases for large scale components having number of components, then these methods will generate huge number of test cases. These formulas are generalized formula, which cannot be extended to large scale.

We propose an efficient method to compute test cases in large scale software. For this purpose we develop a Link-Matrix and on the basis of this link matrix we calculate the number of actual parameters.

A. Link-Matrix

This matrix is composed of rows and columns between components involved in the software. We put the number of parameters 't' (number of parameters between two components is denoted by 't') if there is an integration link between two components and put 0 if there is no interaction between two components.

Link-Matrix can be defined as:

Table 2. Link-Matrix

	C. 1	C. 2	C. 3	C. N	Total
C. 1	0	0 or t	0 or t	0 or t	No. of parameters in 1st row
C. 2	0 or t	0	0 or t	0 or t	No. of parameters in 2nd row
C. 3	0 or t	0 or t	0	0 or t	No. of parameters in 3rd row
….	….	…	…		**…**
C. N	0 or t	0 or t	0 or t	0	No. of parameters in Nth row
Total number of parameters in the Link-Matrix					Total number of parameters in the Link-Matrix

On the basis of Link-Matrix we compute the total number of interaction parameters. Total number of interaction parameters can be computed by adding the number of parameters row wise. Here it is essential to consider that we calculate number of parameters between two components only once. If there is a link between components C1-C2 is computed in first row then it is not taken into account during computing link between C2-C1 in second row.

Now we can compute minimized number of test cases as-

4 * Total number of parameters in Link-Matrix + 1

To illustrate the above defined formula, we implement a case study.

B. Case Study

In this section we define a case study having 5 components. Structure of each component is defined in the Figure 2.

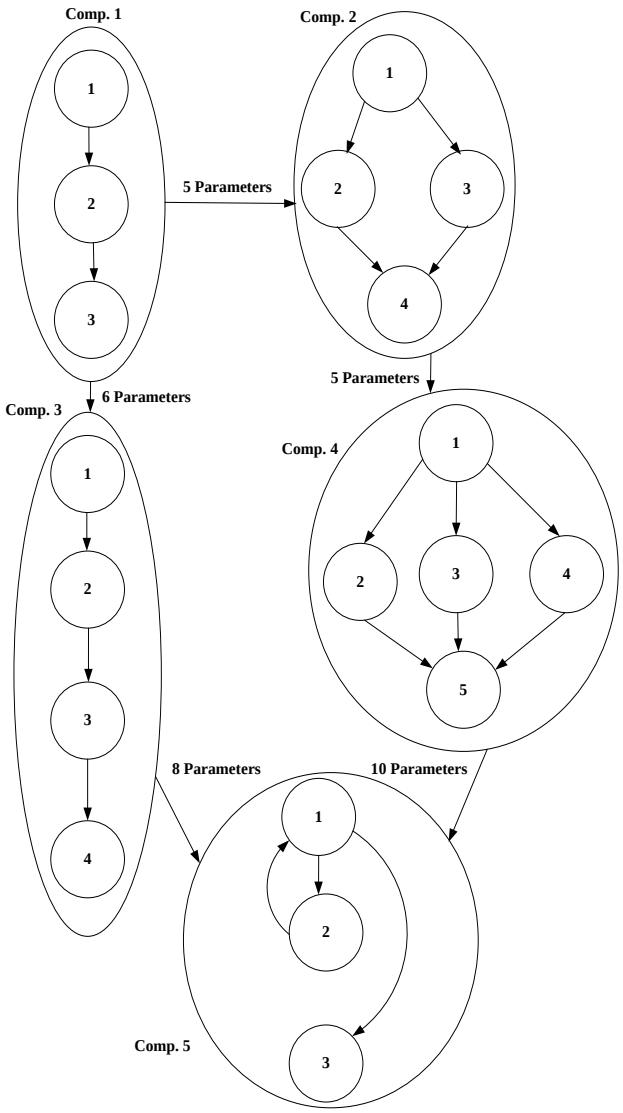

Fig. 2. Black-box Testing

On the basis of Figure, we can draw the Link-matrix as:

Table 3. Link-Matrix

	C.1	C.2	C.3	C.4	C.5	Total
C. 1	0	5	6	0	0	11
C. 2	0	0	0	5	0	5
C. 3	0	0	0	0	8	8
C. 4	0	0	0	0	10	10
C. 5	0	0	0	0	0	0
Total Number of Parameters						**34**

Now we implement the proposed test-case minimization method to compute the total count of test cases. That is,

4 * Total number of parameters in Link-Matrix + 1
= 4 * 34 + 1
= 137

C. Comparison with BVA method

In this segment we make comparison between the proposed method with all three cases of BVA, that is, normal case, robust case and worst case.

i. BVA Normal Case:

Now comparing MNTC with Boundary value analysis method, we get,

Test-case count between components C1 and C2, using BVA

= 4 * 5 + 1 = 21

Test-case count between components C1 and C3, using BVA

= 4 * 6 + 1 = 25

Test-case count between components C2 and C4, using BVA

= 4 * 5 + 1 = 21

Test-case count between components C3 and C5, using BVA

= 4 * 8 + 1 = 33

Test-case count between components C4 and C5, using BVA

= 4 * 10 + 1 = 41

Therefore, total number of test cases of the complete software

= 21 + 25 + 21 + 33 + 41

= 141

Similarly, by applying BVA for robust and worst case, we get following number of test cases, as defined in Table 4.

ii. BVA Robust Case:

Now comparing the proposed method with Boundary value analysis robust case method, we get,

Test-case count between components C1 and C2, using BVA

= 5 * 5 + 1 = 26

Test-case count between components C1 and C3, using BVA

= 5 * 6 + 1 = 31

Test-case count between components C2 and C4, using BVA

= 5 * 5 + 1 = 26

Test-case count between components C3 and C5, using BVA

= 5 * 8 + 1 = 41

Test-case count between components C4 and C5, using BVA

= 5 * 10 + 1 = 51

Therefore, total number of test cases of the complete software

= 26 + 31 + 26 + 41 + 51

= 175

iii. BVA Worst Case:

Now comparing MNTC with Boundary value analysis worst case method, we get,

Test-case count between components C1 and C2, using BVA

$= 5^5 = 3125$

Test-case count between components C1 and C3, using BVA

$= 5^6 = 15625$

Test-case count between components C2 and C4, using BVA

$= 5^5 = 3125$

Test-case count between components C3 and C5, using BVA

$= 5^8 = 390625$

Test-case count between components C4 and C5, using BVA

$= 5^{10} = 9765625$

Therefore, total number of test cases of the complete software

= 3125 + 15625 + 3125 + 390625 + 9765625

= 10178125

Table 4 illustrates the comparative counts among these test case counting methods.

Table 4. Link-Matrix

BVA Normal Case	BVA Normal Case	BVA Worst Case	Proposed MNTC
141	175	10178125	137

IV. CONCLUSION

Testing helps not only in removing bugs and errors from the software but also in enhancing quality assurance, productivity and reusability of software systems [24], [28], [29]. Complex and large software like, component-based software systems are composed of individual, identifiable pieces of code known as components, we need techniques to analyse and to test these components as individual as well as the whole integrated system. We simply can't apply traditional software testing methodologies to test the complex and huge applications. In literature, a large number of software testing techniques are proposed and successfully implemented considering the nature of software development and testing pattern. We have classified two categories of: white-box testing and black-box testing. Proposed minimization technique is very efficient and helpful to minimize the count of test cases. As we can observe from Table 4, that number of test cases achieved using the proposed method are far lesser than traditional BVA method.

REFERENCES

1. G. J. Myers, *The Art of Software Testing.* John Wiley and sons, 2nd Edition, 1979.
2. J. H. Marry, "Testing: A Roadmap, In Future of Software Engineering", in *Proceedings 22nd International Conference on Software Engineering.* June 2000.
3. F. Elberzhager, A. Rosbach, J. Münch, and R. Eschbach, "Reducing Testing Effort: A Systematic Mapping Study on Existing Approaches", Information and Software Technology, vol. 54, no. 10, 2012, pp. 1092–1106.
4. W. T. Tsai, A. Saimi, L. Yu, and R. Paul, "Scenario-Based Object-Oriented Testing Framework", in Proceeding of Third International Conference On Quality Software (QSIC'03), IEEE, 2003, pp. 410 – 417.
5. J. Z. Gao, H. S. Tsao, and Y. Wu, "Testing and Quality Assurance for Component-Based Software", Boston: Artech House, 2003.
6. E. J. Weyuker, Testing Component-Based Software: A Cautionary Tale. IEEE Software. vol. 15, no. 5, 1998, pp. 54-59.
7. C. Ramamoorthy, S. Ho, and W. Chen. "On the automated generation of program test data", IEEE Transactions on Software

Engineering. vol. 2, no. 4, 1976, pp. 293 – 300.
8. J. M. Voas, "A dynamic testing complexity metric", Software Quality Journal. vol. 1, no. 2, 1992, pp. 101 – 114.
9. J. M. Voas, and K. W. Miller, "The revealing power of a test case", Journal of Software Testing, Verification and Reliability, vol. 2, no. 1, 1992, pp. 25 – 42.
10. S. C. Ntafos, "A comparison of some structural testing strategies", IEEE Transactions in Software Engineering. vol. 14, no. 6, 1988, pp. 868 – 874.
11. T. J. Ostrand, and M. J. Balcer, "The category-partition method for specifying and generating functional tests", Communications of the ACM, vol. 31, no. 6, 1988, pp. 676 – 686.
12. J. M. Voas, and K. W. Miller, "Software testability: The new verification", IEEE Software, 1995.
13. E. J. Weyukar, "More experience with data flow testing", IEEE Transactions on Software Engineerin, vol. 19, no. 9, 1993, pp. 912 – 919.
14. L. Michael, R. Sampath and P. A. Aad, "Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development", IEEE Transactions on Reliability. vol. 51, no. 2, 2002.
15. F. Gordon, and A. Andrea, "Whole Test Suite Generation", IEEE Transactions on Software Engineering, vol. 39, no. 2, 2013.
16. A. D. Jehad, and S. Paul, "Generating Class-Based Test Cases for Interface Classes of Object-Oriented Black Box Frameworks", in Proceedings of World Academy of Science, Engineering and Technology. 16, ISSN 1307-6884, 2006.
17. U. K. Tiwari, and K. Santosh, "Components Integration-Effect Graph: A Black Box Testing and Test Case Generation Technique for Component-Based Software", International Journal of Systems Assurance Engineering and Management, Springer, 2016.
18. T. McCabe, "A complexity measure", IEEE Transactions on Software Engineeringi, vol. 2, no. 8, 1976, pp. 308–320.
19. C. Jianguo, "Complexity metrics for component-based software systems", International Journal of Digital Content Technology and its Applications. vol. 5, no. 3, 2011, pp. 235–244.
20. B. Henderson-Sellers, and D. Tegarden, "The application of cyclomatic complexity to multiple entry/exit modules", Center for Information Technology Research Report No. 60, 1993.
21. U. K. Tiwari, K. Santosh, "Cyclomatic complexity Metric for Component Based Software", ACM SIGSOFT Software Engineering Note,. vol. 39, no. 1, 2014.
22. A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, "Using component meta contents to support the regression testing of component-based software", in Proceedings of International Conference on Software Maintenance, Florence, Italy. 2001, pp. 716-725.
23. L. Y. Bixin, Y. W. Zhou, and M. Junhui, "Matrix Based Component Dependence Representation and Its Applications in Software Quality Assurance", ACM SIGPLAN Notices, vol. 40, no. 11, 2005, pp. 29-36..
24. S. G. Nasib, and T. Pradeep, "CBS Testing Requirements and Test Case Process Documentation Revisited", ACM Sigsoft, Software Engineering Notes, vol. 32, no. 2, 2007.
25. S. Tamal, and M. Rajib, "State-Model-Based Regression Test Reduction for Component-Based Software", International Scholarly Research Network ISRN Software Engineering, Article ID 561502.
26. H. E. Stephen, "Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential", Software Testing, Verification and Reliability, vol. 10, no. 4, pp. 249-262.
27. C. Herv´e, and M. M. P. Leonardo, "Exception Handlers for Healing Component-Based Systems", ACM Transactions on Software Engineering and Methodology, vol. 22, no. 4, Article 30, 2013.
28. S. Clemenst, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1997.
29. A. W. Brown, and K. C. Wallnau, "The Current State of CBSE", IEEE Software, 5, 1998.

AUTHORS PROFILE

Dr. Umesh Kumar Tiwari is working as an Associate Professor in Department of Computer Science and Engineering in Graphic Era Deemed to be University, Dehradun. He had received his Ph.D. in 2016. He has more than 12 years of experience in teaching/research of UG and PG level degree courses as a Lecturer/Assistant Professor/ Associate Professor in various academic/research organizations. He is supervisor of 02 PhD and 5 M. tech students who are working on specific domains of software engineering and network security. His research is on multidisciplinary topics and he has published 17 journal papers in reputable international and national journals, and 12 conference papers in reputed international and national conferences. His research interests are Wireless Communication Networks, Network Security, and Software Engineering topics with improved modeling, interaction-integration complexities, testing and reliability models.

Dr. Santosh Kumar had received his Ph.D. from IIT Roorkee (India) in 2012, M. Tech. (CSE) from Aligarh Muslim University, Aligarh (India)in 2007 and B.E. (IT) from C.C.S. University, Meerut (India) in 2003. He has more than 13 years of experience in teaching/research of UG (B. Tech.) and PG (M.Tech.) level courses as a Lecturer/Assistant Professor/ Associate Professor in various academic /research organizations. He has supervised 01 Ph.D. Thesis, 20 M.Tech. Thesis, 18 B.Tech projects and presently mentoring 06 Ph.D students, 03 M.Tech students and 04 B.Tech. students. He has also completed a consultancy project titled "MANET Architecture Design for Tactical Radios" of DRDO, Dehradun in between 2009-2011.He is an active reviewer board member in various national/International Journals and Conferences. He has memberships of ACM (Senior Member), IEEE, IAENG, ACEEE, ISOC (USA) and contributed more than 46 research papers in National and International Journals/conferences in the field of Wireless Communication Networks, Mobile Computing and Grid Computing and software Engineering. Currently holding position of Associate professor in the Graphic Era Deemed to be University, Dehradun (India). His research interest includes Wireless Networks, MANET, WSN, IoT, and Software Engineering.