IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

ANALYSIS OF COMBINATORIAL TESTING

Chaitanya G[1] , Chaitrashree V Bhat[2] , Chandan Kumar R[3] , Shrinitha Bhat[4], Rekha Jayaram[5]

1,2,3,4 Students, Department of Information Science and Engineering, Dayananda sagar College of Engineering
Bangalore, Karnataka, India
5 Associate Professor, Department of Information Science and Engineering, Dayananda sagar College of Engineering Bangalore, Karnataka,
India

Abstract: According to the Studies performed, the combinatorial testing (CT) is an effective approach for detecting faults in the software systems. Combinatorial testing points to identifying faults that arise due to the interaction of values of a small number of inputs. CT technique considerably reduces the testing cost and increases the software system quality. CT also known as t-way testing, reduces the size of the test set by selecting a minimal set of test cases that cover all the viable t-way tuples. An optimal value of t (degree of interaction) for t-way testing for the system would maximize fault detection count in a minimal number of test cases. However, identifying the optimal value of t-way testing for the system remains an open issue. An analysis on different types of combinatorial testing techniques is carried out and a case study on identifying the interaction that exists in the source code, which reduces the count of interactions to be tested using dd path and data flow techniques are discussed. The experimental result indicates that the proposed approach remarkably reduces the count of interactions to be tested without the loss of fault detection capability. The approach can be extended to automate and develop a tool to support the proposed approach .

Index Terms **- Combinatorial testing, t-way testing, Interaction faults, Data flow techniques, DD path graph, Interaction testing.**

I. INTRODUCTION

Software testing is an expensive and time consuming activity that leads to production of reliable software systems [3]. Hence, testing process is allocated a large share of the software development process [3]. However, it is often observed that when the usage of large data-intensive software increases, some of the modules start developing undetected errors even after passing through conventional testing methods, those rare combination of values which have escaped testing process may cause interaction failures.. To avoid those failures, it is recommended to test all combinations of values. However, testing all the combination of values is not feasible either due to time or resources availability. Hence a technique is required that focuses on testing combination of values. Combinatorial testing could significantly reduce test cost and increase quality of software system. It has been proved to be effective especially in a software system where faults come from the interactions of its parameters.

According to the observation, Combinatorial testing produces large amount of faults are caused by few input parameter interactions. Hence instead of testing all combinations of values, combinations of only few parameters are tested. In order to generate test set, values for input parameters are selected such that every possible combination of values of any t parameters occurs at least once, t is interaction strength.

In this paper three different combinatorial testing techniques are discussed and a novel approach is presented that presents the interactions of variables that exist in the code. We have extended an existing work [3] by modifying their approach to handle the Object Oriented Programs. The previous approach to identify the interactions among variables was meant for structured programs where the program is divided into modules known as functions. A data flow graph is generated from the source code and data flow technique is applied on it. The c-use of a variable is redefined for the proposed approach. From the obtained flow graph and usage of variables, interactions are identified. This approach achieves a remarkable reduction in the count of interactions to be tested. As a result, instead of testing all t-way interactions, only the identified interactions are tested which leads to reduction in testing costs.

II. RELATED WORK

Combinatorial testing (CT) has been widely studied and has become a well-accepted testing method. A large number of research articles have focused on CT.

According to research in [1] we can reduce the cost of testing by reusing the test oracle through "combinatorial join".
Test oracle is the important part of test case. It defines the expected output that satisfies the software specification. The actual output is checked against the test oracle to determine whether a test case is passed or failed.
Generation of test oracle is time consuming and costly process. Sometimes generation of test cases involves experts especially when the program contains complex functions.
Their research gives a novel approach called "combinatorial join" which helps us to join two covering arrays into one. Using this approach test oracle used in unit testing phase can be reused in integration testing phase.
For example , consider two covering arrays generated for two different components A and B, both A and B has three factors(i.e., *A*1, *A*2, *A*3, and *B*1, *B*2, *B*3) with only two possible values(1,2) as shown in figure 1. J represents the covering array generated by combinatorial join.

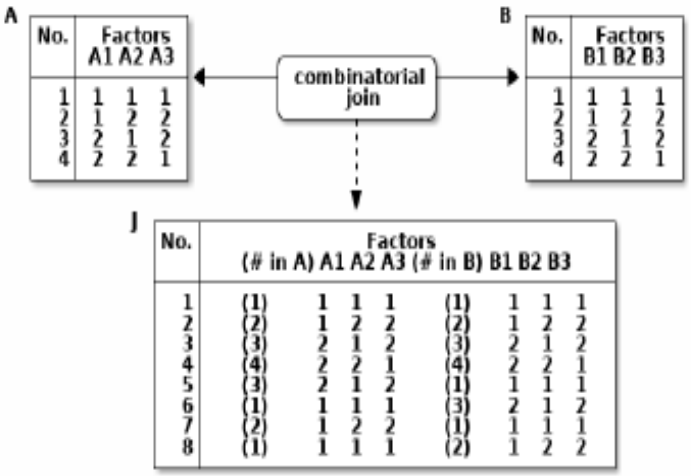

Fig 1. Example for combinatorial join

Rather than generating a new covering array for six factors ,this approach called combinatorial join generates a new covering array *J* by selecting rows from existing covering arrays (*A* and *B*).

This paper not only works on joining two covering array but also tries to minimise the size of new covering array generated.
According to research this approach can reduce the testing cost by 55- 95%.This approach have been implemented in JCUnit ,an automated CIT tool built on JUnit2. The tool constructs a covering array by applying "divide and-conquer" approach. This approach best fits with combinatorial testing to generate test oracle.

According to the research in [2] they presented an empirical study of CT on five industrial systems with real faults. The details of the input space model (ISM) construction, such as factor identification and value assignment, were included. Comparison was made between the faults detected by CT with those detected by the in-house testing teams using other methods, and the results suggested that despite some challenges, CT is an effective technique to detect real faults, especially multi-factor faults, of software systems. A common approach is to treat each function of the system as a SUT (System Under Test). For each SUT identified, requirements are interpreted to construct the corresponding ISM using several techniques, such as equivalence partitioning, boundary values analysis, and randomization. Once the ISM is constructed, ACTS used to generate a CT test suite. Depending on the difficulty of the test execution and output verification, input model was refined several times before actual test execution. This was done to potentially improve the fault detection results, but keep the testing cost to an acceptable level. It indicated that any test suite that is larger than this size is likely to require too much time for test execution and output verification. As a result, only 2-way test suites were used in case studies. Once test execution and output verification, is finished then compared the faults detected by CT with those detected by the in-house testing team.

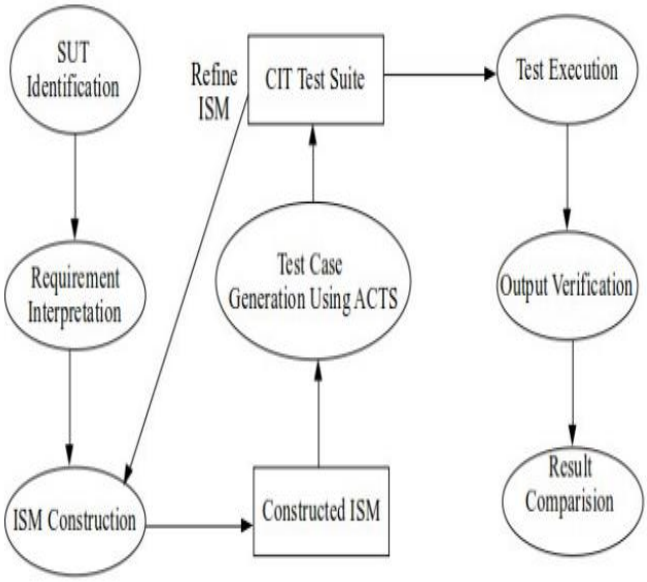

Fig 2.An overview of experimental design

According to the research in [3] number of interactions to be tested using combinatorial testing can be reduced. Here concept of path testing is combined with combinatorial testing. The approach uses data flow concepts to compute the usage of variables to identify interactions among variables.

Here a control flow graph is implemented from the source code which is then converted into DD path graph.DD path graph is further reduced such that complexity of DD path and Reduced DD path graph remains the same. Certain rules have been defined to reduce the DD graph which is shown in Step C of case study . Usage like c-use, p-use and r-use are computed at both statement level and block level.

This step is followed by identification of interactions that causes type1 interaction failures and type2 interaction failures. Where type1 interaction failure are caused by set of variables that are p-used in a path and type2 interaction failure are caused by that are c-used in a path.

By the end of this step, a set of probable interactions to cause type 1 interaction failures and a set of probable interactions to cause type 2 interaction failures is obtained.

Final step is finding minimum set of interaction. The union of these two sets results into a minimal set of interactions that exist in the source code.

Results of this research indicate that this approach significantly reduces the count of interactions to be tested without significant loss of fault detection capability.

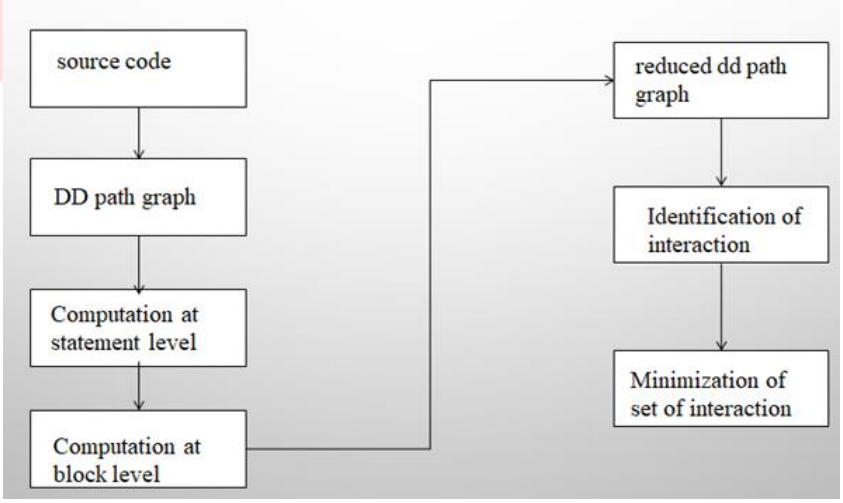

Fig 3. Flow chart of research paper [3]

III. CASE STUDY

In order to demonstrate the effectiveness of the approach proposed in [3], case study has been used. Case study is a billing application program in which admin can list, modify and delete the products and customer can get his items billed.

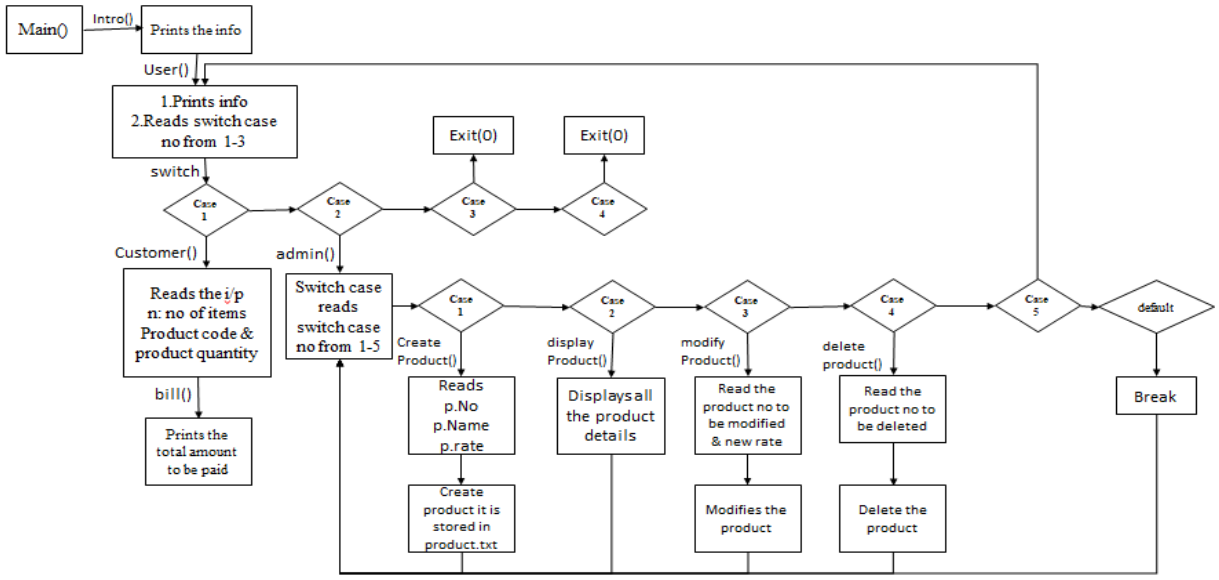

Fig 4.Shows the algorithm for case study

Appendix contains source code for case study.
Step A: The source code is converted into DD path.
This process can automated as shown in research [4]

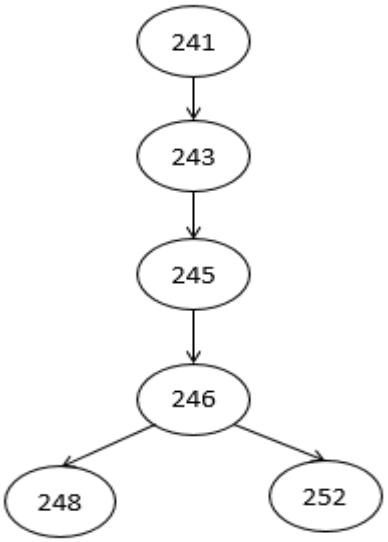

Fig 5(a) shows an example of DD path for createProduct( ) function .

Step B: At each statement, the c-use and p-use of the variables and ruse of the functions are identified and are shown below. r-use is ignored because none of the returned value is c-used or p-used.
user( ) :
Statement 145 → ch (c-use) ,Statement 146 → ch (p-use)
admin( ):
Statement 222 → ch (c-use) , Statement 223 → ch (p-use)
createProduct( ):
Statement 241 → n (c-use) , Statement 143 → p (c-use), Statement 245 → r (c-use)
Product(n, p, r):
Statement 21 → productNumber (c-use), Statement 22 → productName (c-use), Statement 23 → productRate (c-use)
notPresent( ):
Statement 200 → n (p-use and c-use), Statement 202 → p ,r (c-use), Statement 203 → n,x,p,y (p-use)
displayProduct( ):
Statement 263 → n (c-use, p-use), Statement 265 → p, r (c-use)
modifyProduct( ):

Statement 283→ modnum (c-use), Statement 289 → isUpdated (c-use), Statement 290 → n (c-use,p-use), Statement 295 → p, r (c-use), Statement 296 → n, modnum (p-use), Statement 299 → r (c-use), Statement 301 → isUpdated (c-use), Statement 315 → isUpdated (p-use).
deleteProduct( ):
Statement 327→ modnum (c-use), Statement 333 → isDeleted (c-use), Statement 334 → n (c-use ,p-use), Statement 336 → p, r (c-use), Statement 337 → n, modnum (p-use), Statement 347 → n, modnum (p-use), Statement 348 → isDeleted (c-use), Statement 354 → isDeleted (p-use).
customer( ):
Statement 159→ n (c-use), Statement 163 → i (c-use ,p-use),n (p-use), Statement 167 → code (c-use), Statement 168 → z (c-use), Statement 169 → z (p-use), Statement 171 → qty (c-use), Statement 176 → i (c-use, p-use),n(p-use), Statement 178 → totalAmount (c-use), Statement 183 → i (c-use,p-use),n(p-use).
getItem( ):
Statement 86 → isFound (c-use), Statement 87 → pn (c-use,p-use), Statement 89 → pp,pr (c-use), Statement 90 → pn, code (p-use), Statement 99 → isFound (c-use).
setQuantity( ):
Statement 108 → quantity (c-use).
getAmount( ):
Statement 113 → r (c-use).
setNumber(pn):
Statement 50 → productNumber (c-use).
setName(pp):
Statement 54 → productName (c-use).
setRate(pr):
Statement 58 → productRate (c-use).

Step C: The graph is transformed to an RDD path graph.

Aim of this step is to reduce the count of nodes present in DD path graph. A node where no usage is present can be removed .Elimination is done such that complexity of both DD path graph and RDD path graph remains the same and structure of the graph is not modified.
The in-degree of a node is defined as the number of edges entering a node whereas out-degree represents number of edges leaving a node.

If the in-degree of a node is 0 and out-degree is 1, then node is called start node. If the in-degree of a node is 1 and out-degree is 0, then node is called end node. If the in-degree of node is greater than 1 and out-degree is 1, then such node is called merge node. If in-degree of node is 1 and out-degree of node is greater than it is called branch node.

Reduction Rules for DD path graph:

1. If the in-degree and the out-degree of a node are 1, then the node can be removed by creating an edge by joining its predecessor node to its successor node.
2. If a node is start node and there is no usage, then such a node can be removed.
3. If a node is end node and there is no usage, then such a node can be removed.
4. If the node is a merge node then, that node can be removed by connecting all the incoming edges of the node to its successor node.

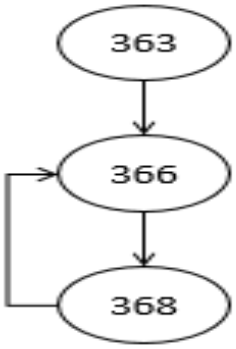

Fig 6(a) shows RDD path for main( )function

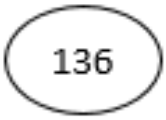

Fig 6(b) shows RDD path for intro( )function

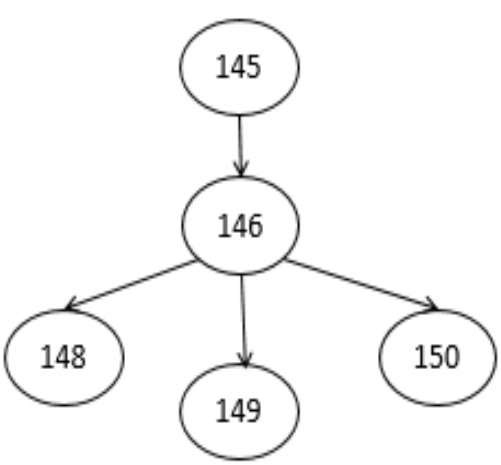

Fig 6(c) shows RDD path for user( )function

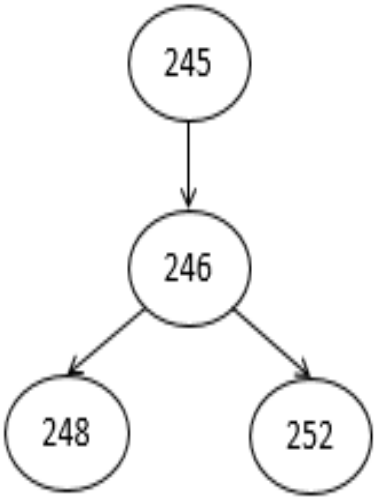

Fig 6(e) shows RDD path for createProduct( )function

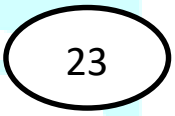

Fig 6(f) shows RDD path for product( )function

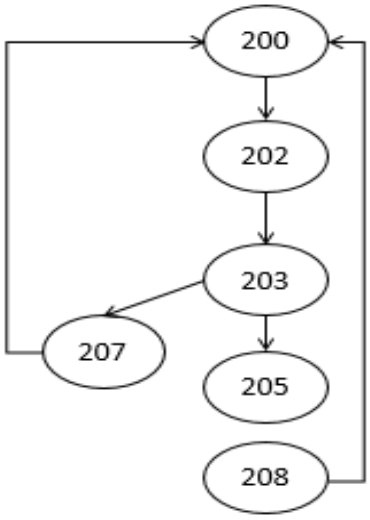

Fig 6(g)shows RDD path for notPresent( )function

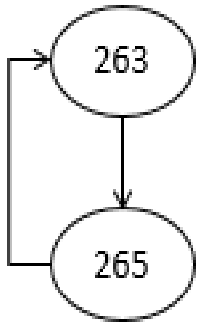

Fig 6(h)shows RDD path for displayProduct( ) function

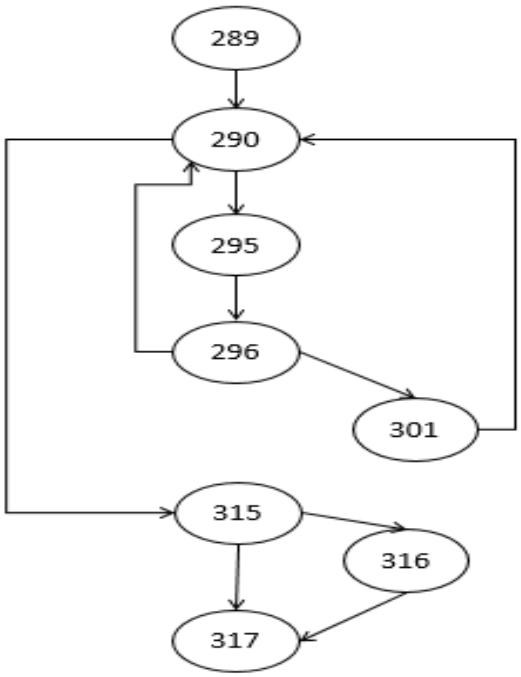

Fig 6(i) shows RDD path for modifyProduct( ) function

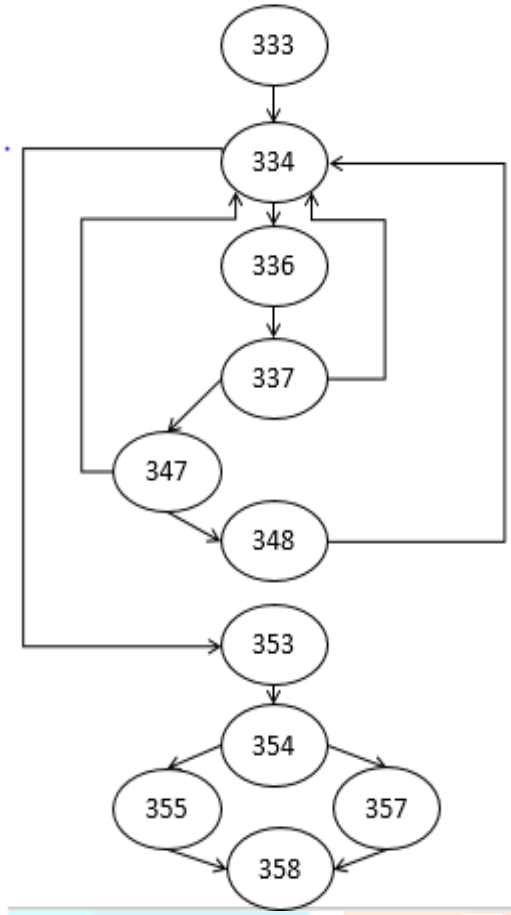

Fig 6(j) shows RDD path for deleteProduct( ) function

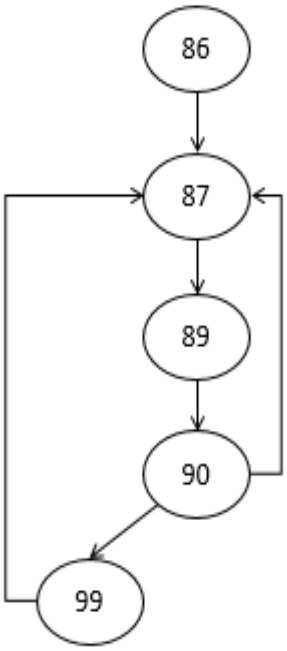

Fig 6(l) shows RDD path for getItem( )function

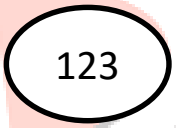

Fig 6(m) shows RDD path for printItemDet()function

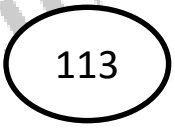

Fig 6(n) shows RDD path for getAmount( ) function.

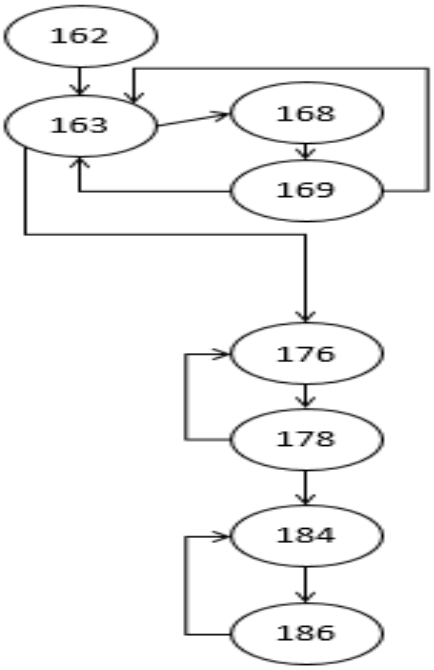

Fig 6(k) shows DD path for customer( )function

DD paths graph for functions like main( ),intro( ), getname() cannot be reduced.

Rather than working on a DD path graph having large number of nodes, we work on RDD path graph having

less number of nodes .

Step D: At each block, the c-use and p-use of the variables and ruse of the functions are identified and are shown in Table 2.
user( ) :
Node 145 → ch (c-use) ,Node 146 → ch (p-use)
admin( ):
Node 222 → ch (c-use) , Node 223 → ch (p-use)
createProduct( ):
Node 245 → n, p, r (c-use).
Product(n, p, r):
Node 23 →productNumber, productName, productRate
(c-use).
notPresent( ):
Node 200 → n (c-use, p-use), Node 202 → p, r (c-use), Node 203 → n, x, p, y (p-use).
displayProduct( ):
Node 263 → n (c-use,p-use), Node 265 → p, r (c-use).
modifyProduct( ):
Node 289 → modnum,isUpdated (c-use), Node 290 → n (c-use, p-use), Node 295 → p, r (c-use), Node 296 → n, modnum (p-use), Node 301 → r, isUpdated (c-use), Node 315 → isUdated (p-use).
deleteProduct( ):
Node 333 → modnum,isDeleted (c-use), Node 334 → n (c-use, p-use), Node 336 → p, r (c-use), Node 337 → n, modnum (p-use), Node 347 → n, modnum (p-use), Node 348 → isDeleted (c-use), Node 354 → isDeleted (p-use).
customer( ):
Node 162 → n (c-use), Node 163 → i (c-use,p-use),n(p-use), Node 168 → code, z(c-use), Node 169 → z(p-use), Node 176 → qty,i(c-use), i,n(c-use,p-use), Node 178 → totalAmount(c-use), Node 184 → i(c-use,p-use),n (p-use).
getItem( ):
Node 86 → isFound (c-use), Node 87 → pn (c-use,p-use), Node 89 → pp,pr (c-use), Node 90 → pn, code (p-use), Node 99 → isFound (c-use).
setQuantity( ):
Node 108 → quantity (c-use).
getAmount( ):
Node 113 → r (c-use).
setNumber(pn):
Node 50 → productNumber (c-use).
setName(pp):
Node 54 → productName (c-use).
setRate(pr):
Node 58 → productRate (c-use).

Step E: Identification of interactions
From the RDD path graph, independent paths are computed and probable interactions to cause type 1 interaction failures are identified. To identify interactions that can cause type 1interaction failures, p-use of the variables at each node of the graph is considered.
notPresent() has complexity of 3 ,independent path identified are 200-202-203-205, 200-202-203-207-208, 200-202-203-207-200-202-203-205 which gives {n , p , x , y } interaction. Similarly ,function modifyProduct(),which has complexity of 4 has {n , modnum , isUpdated} interaction, deleteProduct( ) has complexity of 5 leads to {n, modum, isDeleted} interaction, customer( ) has complexity of 5 and it leads to {i , n, z} interactions .getItem ( ) leads to { pn, code} interaction which has complexity of 3.
Interactions among variables that may cause type 2 interaction failures in case study.
From the RDD path graph, independent paths are computed and probable interactions to cause type 2 interaction failures are identified. To identify interactions that can cause type 2 interaction failures, c-use of the variables at each node of the graph is considered.

notPresent() has complexity of 3 ,independent path identified are 200-202-203-205, 200-202-203-207-208 , 200-202-203-207-200-202-203-205 which gives {n , p , r } interaction.
Similarly ,function modifyProduct(),which has complexity of 4 has {n , p ,r , modnum , isUpdated} interaction, deleteProduct( ) has complexity of 5 leads to {n , p , r , modum, isDeleted} interaction, customer( ) has complexity of 5 and it leads to {i , n, z, code ,qty ,totalamount } interactions .getItem ( ) leads to { pn , pp , pr , code } interaction which has complexity of 3.createProduct () with complexity of 4 has { n , p , r } interaction. Product() has only 1 path aand it leads to{productName , productNumber , productRate}interaction.

Step F: Minimizing the set of interactions
{n , modnum , isUpdated} ,{n , modnum, isDeleted} ,{n , p , x , y } ,{i , n, z} ,{ pn, code} there are the interactions that may lead to type 1 interaction failure . { n ,p , r , modnum , isUpdated},{n, p , r , modum, isDeleted},{i , n, z , code ,qty ,totalamount },{ pn ,

pp , pr , code} are variables which may lead to type 2 interaction failures. We can observe that variable like isUpdated, isDeleted, z are flag variables hence can be ignored. Table 5 and 6 shows the final minimum set.

Table 1: Interactions among variables that may cause type 1 interaction failures in case study.

Interactions
{n , modnum }
{n , modum }
{n , p , x , y }
{ pn, code}

Table 2: Interactions among variables that may cause type 2 interaction failures in case study.

Interactions
{ n ,p , r , modnum }
{n, p , r , modnum}
{code ,qty ,totalamount }
{ pn , pp , pr , code}

Note : {n ,modnum}and {n, p , r , modnum} is written twice as variable modnum is used twice in the program ,first usage is in modifyProduct( ) and second is in deleteProduct( ). Both of them leads to different testcases.

Further, test cases will be written to exercise these variables rather than testing all the combination of input parameters.

IV. CONCLUSION

Based on case study we can conclude that we can minimize the number of interactions to be tested.We can see that case study has inputs parameters like ProductName, ProductNumber, ProductRate, ModifyNumber, new rate,DeleteNumber which is given by admin and inputs like number of items , code and quantity are given by customers. Rather than testing all the combinations of these input parameter ,we can test only those interaction mentioned in table.5 and 6 by using only 78 testcases which is far less than testcases generated using conventional way.78 testcases are generated based on constrains like products present in market is only 3 and customer can buy within those limitations. Therefore the approach presented in research significantly reduces the count of interactions to be tested without significant loss of fault detection capability.

V. ACKNOWLEDGMENT

We express sincere thanks to Prof. C P S Prakash, Principal Dayanada Sagar College Of Engineering. We would like to extend our gratitude to our HOD RamMohan Babu, mentor Dr Suma and to our guide Prof. Rekha jayaram Dayanand Sagar College of Engineering, for co-operating and guiding us through the process.

APPENDIX

(1-5) //code initialization
(6-11) //variable declaration
(12-18)Product()//default constructor called
(19-20)Product(int pNum, string pNam, float
rate)//: productNumber(pNum), productName(pNam), productRate(rate)
(21) productNumber = pNum;
(22) productName = pNam;
(23) productRate = rate;
(24-32) //opening text file to enter productNumber, productName, productRate
(33-35) //product created successfully
(36) int getNumber()
(38-39) return productNumber;
(40) string getName()
(42-43) return productName;
(44) float getRate()
(46-47) return productRate;
(48) void setNumber(int x)
(50-51) productNumber = x;
(52) void setName(string x)
(54-55)productName = x;
(56) void setRate(float x)
(58-60) productRate = x;
(61-76) //creating bill class and displaying date,time
(77-78) int quantity = 0;

(79) int getItem(int code)
(80-85)//declaring variables
(86) bool isfound = true;
(87) while (prod>>pn)
(88-89)prod >> pp >> pr;
(90) if (pn == code)
(91-96)//set name,number,rate and return 1
(97) else
(98-99) isfound = false;
(100-105)//out of if print product does not exits and return 0
(106) void setQuantity(int x)
(107-109)quantity = x;
(110) float getAmount()
(111-112) //declaring variable
(113) r = product.getRate();
(114-115) return r*quantity;
(116) void printItemDet()
(117-124)//printing details of product
(132) static void intro()
(133-136) //billing system
(137) static void user()
(138-144) //select option from customer, administrator, exit
(145) cin >> ch;
(146) switch (ch)
(148) case 1: customer(); break;
(149) case 2: admin(); break;
(150) case 3: exit(0);
(151) default:cout << "choose correct number"; break;
(152-153) end
(154) static void customer()
(155-158) //enter number of items
(159) cin >> n;
(160) displayProduct();//display all products
(161-162) //declaring variables
(163) for (int i = 0; i < n; i++)
(164-166)//enter product number
(167) cin >> code;
(168) int z = item[i].getItem(code);
(169-170) if (z){
(171) cin >> qty;
(172) item[i].setQuantity(qty);
(173-175)//declaring variables
(176) for (int i = 0; i < n; i++)
(177-182)fBill.totalAmount += item[i].getAmount();
(183) for (int i = 0; i < n; i++)
(184-192)//print total amount and thank you message
(193)static int notpresent(int x,string y)
(194-199) //declaring variables n,p,r
(200) while (notpresent >> n)
(201-202) notpresent >> p >> r;
(203) if (n == x|| p==y)
(204-207) return 0;
(208-209)//out of while loop and if block return 1;
(210) static void admin()
(212) while (1)
(213-221)//variable declaration
(222) cin >> ch;
(223-224) switch (ch)
(225)case 1:createProduct(); break;
(226)case 2:displayProduct(); break;
(227)case 3:modifyProduct(); break;
(228)case 4:deleteProduct(); break;
(229)case 5: user(); break;
(230) default: break;
(231-233)end
(234)static void createProduct()
(235-240) //declaration of variables to enter product details
(241) cin >> n;

(243)cin >> p;
(245)cin >> r;
(246)if (notpresent(n,p))
(248) Product prod(n, p, r);
(250) else
(251-254) //print product already exits
(255) static void displayProduct()
(256-262) //declaration of variables
(263) while (displayProducts >> n)
(265)displayProducts >> p >> r;
(266-274) //displaying on screen
(275)static void modifyProduct()
(276-277)displayProduct();//first display all products for the users to select the product
(278-282)//enter product number to modify
(283)cin >> modNum;
(284-288)//declaring variables
(289) bool isUpdated = false;
(290) while (prodIn >> n)
(291-294) //comments
(295) prodIn >> p >> r;
(296) if (n == modNum)
(297-298)//enter new rate
(299) cin >> r;
(300-302)isUpdated=true;//product successfully updated
(303-314) //closing text file after update
(315) if (!isUpdated)
(316-318) //product number not found
(319) static void deleteProduct()
(320-321) displayProduct();//first display all products for the users to select the product
(322-326) //Enter the Product Number to Delete
(327) cin >> modNum;
(328-332) //declaring variables
(333) bool isDeleted = false;
(334) while (prodIn >> n)
(335-336) prodIn >> p >> r;
(337) if (n != modNum)
(338-346) consider same n,p,r value to prodTmp
(347) else if (n == modNum)
(348) isDeleted = true;
(349-353) //closing text file after deleting
(354) if (isDeleted)
(355) cout << "\n\n\n\tPRODUCT DELETED SUCCESSFULLY";
(356) else
(357) cout << "\n\nPRODUCT CODE NOT FOUND";
(358-360) end
(361) int main()
(362-365) //call intro() function
(366) while (1)
(367-370) //call user() function

REFERENCES

[1] Hiroshi Ukai Rakuten, Xiao Qu, Hironori Washizaki and Yoshiaki Fukaza," Reduce Test Cost by Reusing Test Oracles through Combinatorial Join", 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (CSTW).
[2] Linghuan Hu & W. Eric Wong & D. Richard Kuhn & Raghu N. Kacker,"How does combinatorial testing perform in the real world: an empirical study", published on 20 April 2020,Springer Science+Business Media, LLC, part of Springer Nature 2020.
[3] Sangeeta Sabharwal, Manui Aggarwal," A novel approach for deriving interactions for combinatorial testing ",Engineering Science and Technology, an Intemational Journal 20 (2017) 59-71.
[4] Hina Sattar ,Imran Sarwar Bajwa,Umar Farooq Shafi,Ikram Ul Haq, **"**Automated DD-path testing: A challenging task in software testing**",** 9th International Conference on Digital Information Management, ICDIM 2014
[5] Wei Zheng, Xiaoxuc Wu, Desheng Hu, and Qihai Zhu," Locating Minimal Fault Interaction in Combinatorial Testing", Hindawi Publishing Corporation Advances in Software Engineering ,volume 2016.
[6] Paul C. Jorgensen's Software Testing, A Craftsman's Approach Fourth Edition.