Volume 5, Issue 7, July 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Test Case Prioritization Based on Data Flow Testing

Priyanka*, Harish Kumar, Naresh Chauhan
Department of Computer Engineering,
YMCA University of Science & Tech.,
Haryana, India

Abstract— Data-flow testing monitors the lifecycle of a piece of data and looks out for inappropriate usage of data during definition, use in predicates, computations and termination (killing). It identifies potential bugs by examining the patterns in which that piece of data is used. While identifying the test cases of data flow testing, there may be large number of test cases. It may not be possible for a tester to execute all the test cases identified in this huge test suite due to time and cost constraints. Therefore, there is need to prioritize the test cases so that the important test cases are executed first that identify the critical bugs earlier. To address this problem the concept of du(def–usage) and dc(def-clear)-paths has been taken. So in this paper, we proposed a test case prioritization technique based on du and dc-paths.

Keywords— Data flow testing, du and dc-paths, test case prioritization, APFD

I. INTRODUCTION

Defects can exist in the software, as it is developed by human beings who can make mistakes during the development of software. However, it is the primary duty of a software vendor to ensure that software delivered does not have defects and the customers' day-to-day operations do not get affected. This can be achieved by rigorously testing the software. Different types of testing approaches exist such as black box testing, white box testing and grey box testing. For the purpose of doing static and dynamic testing, various techniques are followed. System testing is also one of major type. In it load testing [1], stress testing and volume testing are used for checking performance, recovery testing to check recovery point, configuration testing to check various configuration on the system and regression testing [2, 36, 47] for the purpose of revalidation of the new versions of the existing software and many more.

Regression testing [3] as the name suggests is used to test / check the effect of changes made in the code. Ideally, Regression testing executes the existing as well as additional test cases due to changes. However, this is impractical to execute all the test cases due to constraints of time and cost of the project. Most of the time the testing team is asked to check last minute changes in the code just before making a release to the client, in this situation the testing team needs to check only the affected areas.

It is impractical and inefficient to re-execute every test for every program function once a change has occurred. Regression tests should follow on critical module function. To make regression testing easier, software engineers typically reuse test suites of the original program, but also new test cases may be required to test new functionalities of the new version. The focus here is on the reuse of test cases as most ideas about costs and benefits come from test suite granularity. There are four methodologies, considered here, that reuse the test suites of the original version of the software: *retest-all*, *regression test selection (RTS) [4,12]*, *test suite reduction (TSR)* and *test case prioritization (TCP)[5,6,13,14,15]*. Retest all, reruns every test case of the test suite. It is not a feasible approach as time period to complete the work is fixed. RTS, on the other hand selects some of the test cases from the test suite on temporary basis where as TSR permanently removes test cases from the test suite depending upon the modification in the existing project. Selection and removal of test cases from test suite can be problematic in some situation. So, a new methodology is there namely test case prioritization.

Test case prioritization techniques arrange test cases so that those test cases that are better at achieving the testing objectives are run earlier in the regression testing cycle [22]. For instance, software engineers might want to schedule test cases so that code coverage is achieved as quickly as possible or increase the possibility of fault detection [42, 44] early on in the testing. The improved rates of fault detection can provide early feedback on the software being tested. This reduces regression testing costs by allowing the software engineers to tackle the discovered faults and begin debugging earlier in testing than what might otherwise be possible.

Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors. Errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. The data usage for a variable affects the white box testing and thereby the regression testing. If the prioritization of regression test suite is based on this concept, the rate of detection of faults will be high and critical bugs can be discovered earlier. Therefore, in this research paper, du-paths which are variable usage paths and the paths wherein the variable is defined more than once called non-dc paths (non-definition clear paths) have been taken for the test cases prioritization.

The next section, describes background information and related work relevant to test case prioritization techniques. Section III describes the proposed approach. Section IV presents results and analysis, and section V presents conclusions and future work.

II. BACKGROUND AND RELATED WORK

Data-flow testing [7,8] is a white box testing technique that can be used to detect improper use of data values due to coding errors. Errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. Additionally, he/she may define a variable, but not initialize it and then use that variable in a predicate.

Data-flow testing [7,8] monitors the lifecycle of a piece of data and looks out for inappropriate usage of data during definition, use in predicates, computations and termination (killing). It identifies potential bugs by examining the patterns in which that piece of data is used. For example, A pattern which indicates usage of data in a calculation after it has been killed is certainly a bug which needs to be addressed.

To examine the patterns, we need to construct a control flow graph of the code. A control flow graph is a directed graph where the nodes represent the processing statements like definition, computation and predicates while the edges represent the flow of control between processing statements. Since data-flow testing closely examines the state of the data in the control flow graph, it results in a richer test suite than the one obtained from traditional control flow graph testing strategies like all branch coverage, all statement coverage, etc.

J. Rummel et al [9] proposed an approach to regression test prioritization that leverages the all-DUs test adequacy criterion that focuses on the definition and use of variables within the program under test. This prioritization scheme is based on empirical studies that have shown that test cases fulfilling the all-DUs test adequacy criteria are more likely to reveal defects than those that meet the control flow-based criteria.

Yogesh Singh et. al. [10,11] proposed a hybrid approach using variables that combine both selection and prioritization. It considered source code changes and coverage information with respect to each test case. Variables are vital source of changes in the program, and this method captured the effect of changes in terms of variable computation. This work is focused on the uses and computation of variables, changes in variables may induce new def-use associations. Their work addresses the coverage of the def-use (DU) paths that are not def-clear, using the test cases obtained after implementation of technique.

III. PROPOSED WORK

In data-flow testing, the first step is to model the program as a control flow graph. This helps to identify the control flow information in the program. In step 2, the associations between the definitions and uses of the variables that is needed to be covered in a given coverage criterion is established. In step 3, the test suite is created using a finite number of paths from step 2.

In this method control flow graphs of the programs are prepared first. Then all the du-paths and dc-paths are identified in that flow graph. The test suite is prepared for the programs which covers all the independent paths of the program. After this, the du-paths which are not dc are identified. Then a table is prepared showing the test cases and 'number of non-dc' paths, 'number of dc paths' and 'number of lines of code' covered by each test case. The test suite is prioritized such that the test cases which covers higher number of non-dc paths are executed first. In case two test cases cover same number of non-dc paths then the test case which covers higher number of dc paths are executed first out of those two test cases. Further, if a condition arises that two test cases cover same number of dc paths then the test case which covers the higher number of lines of code is executed first. After making the sequence of these test cases which covers one or more non-dc path in a program, the sequence of remaining test cases is kept random for execution. The exact flow of making the prioritized test suite is shown in Fig1.

Data-flow testing [8] monitors the lifecycle of a piece of data and looks out for inappropriate usage of data during definition, use in predicates, computations and termination (killing). It identifies potential bugs by examining the patterns in which that piece of data is used. While identifying the test cases of data flow testing, there may be large number of test cases. It may not be possible for a tester to execute all the test cases identified in this huge test suite due to time and cost constraints. Therefore, there is need to prioritize the test cases so that the important test cases are executed first that identify the critical bugs earlier. For this purpose the concept of du-and dc-paths has been taken. As discussed in the chapter 2, there are def-use(du) and def-clear(dc) paths in data flow programs. The du-paths which are not dc-paths (non-dc) are supposed to be more problematic from view point of a tester. It means that there may be more bugs in the du-paths which are not dc-paths. So the test cases based on this concept have been prioritized in this work for original programs.

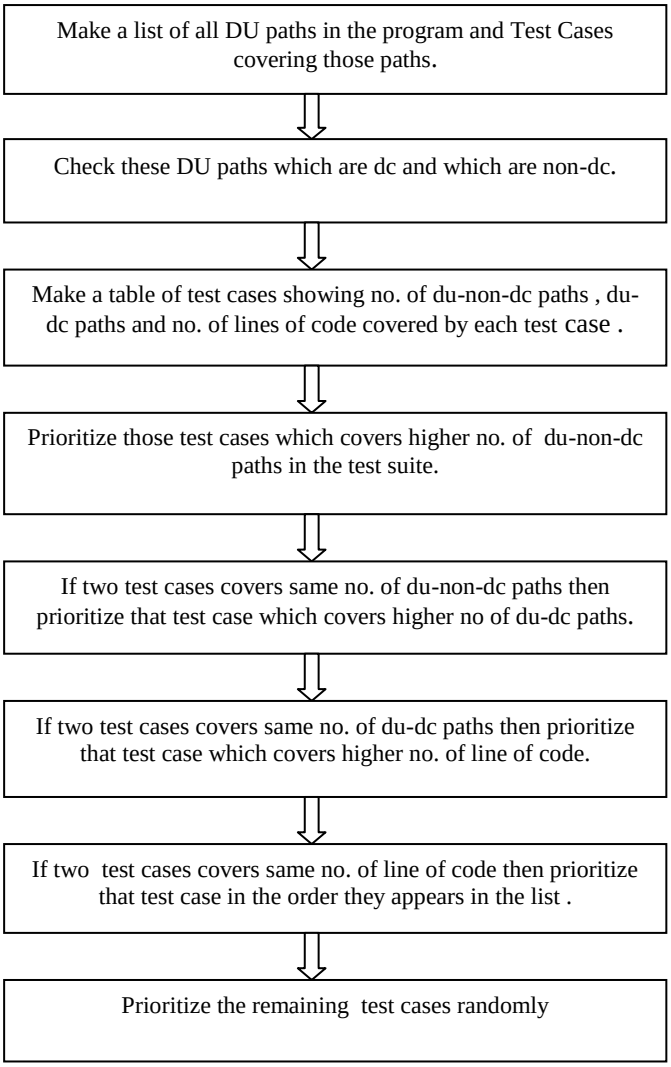

Fig:1.Control flow chart of method to prioritize the test suits

IV. EVALUATION AND ANALYSIS

To analyze and validate the proposed method for prioritizing test cases a C program [8] has been taken. The control flow graph of program was prepared first. Then all du-paths and dc-paths are identified. Before applying the proposed approach some errors have been introduced intentionally in the program. Further, to validate the prioritized test suite, the APFD (Average percentage of fault detection) metric has been used. The validation has been demonstrated by comparing the randomized test suite and the prioritized test suite with the help of APFD graphs. The findings of the proposed approach are shown below.

After analysis of the CFG all independent path are determined. Table 1 shows all independent paths of the sample program.

Table 1 : Independent Path of program

S.NO.	Path No.	Independent Path
1	Path 1	$N_1N_2N_{23}$
2	Path 2	$N_1N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3	Path 3	$N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4	Path 4	$N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5	Path 5	$N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6	Path 6	$N_1N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7	Path 7	$N_1N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8	Path8	$N_1N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

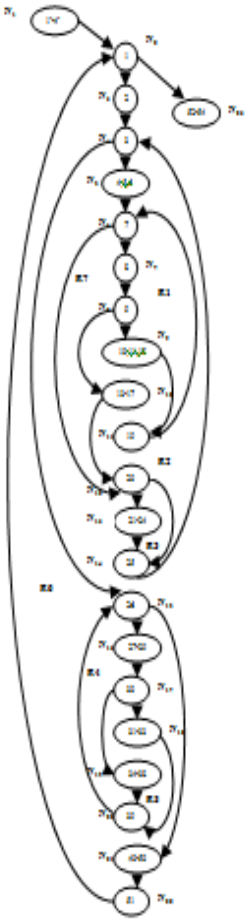

Fig. 1 Control flow graph of the considered program

Table 2:Test Case Design on Program from the list of Independent Paths

Test Case ID	Inputs		Expected Output	Independent path covered by Test Case	Total lines of code covered by test case
	Source	Amount			
1	Agriculture	400000	Source Agriculture Amount 400000 Do you want to enter more(y/n) Y	1), 2), 3), 5), 8)	42 lines
	Others	100000	Source Others Amount 100000 Do you want to enter more(y/n) N		
			Total 500000		
2	1234		Source can contain only character.	1), 2), 3), 4)	19 lines
3	Agriculture and others		Source can contain a max of 20 characters.	1), 2), 3), 6)	24 lines
4	Agriculture	0	Amount cannot be less than or equal to 0	1), 2), 3), 5), 7)	31 lines

Table 3 : Definition Nodes and Usage Nodes in Program

Variable	Defined At	Used At
Source	1', 6	7, 9, 20, 44
Amount	29	30, 45, 46
Total	2', 46	46, 52
Flag1	3',11,16,23, 49	3
Flag2	3', 32, 37, 50	26
I	7	7, 9, 15
Income_ch	4', 48	1

Table 4 : du and dc paths in Program

Variable	du Path(beg-end)	dc?	Test case which covers this path
source	1'-7	No	1,2,3,4
	1'-9	No	1,2,3,4
	1'-20	No	1,2,3,4
	1'-44	No	1
	6-7	Yes	1,2,3
	6-9	Yes	1,2,3
	6-20	Yes	1,2,3
	6-44	Yes	1,
amount	29-30	Yes	1,4
	29-45	Yes	1
	29-46	Yes	1
total	2'-46	Yes	1
	2'-52	No	1
	46-46	Yes	1
	46-52	Yes	1
Flag1	3'-3	Yes	1,2,3,4
	11-3	No	1,3,4
	16-3	No	2
	23-3	Yes	3
	49-3	Yes	Not a possible path
Flag2	3'-26	Yes	1,4
	32-26	Yes	1
	37-26	Yes	4
	50-26	Yes	1
i	7-7	Yes	1,2,3,4
	7-9	Yes	1,2,3,4
	7-15	Yes	2
Income_ch	4'-1	Yes	1,2,3,4
	48-1	Yes	1

There are total 09 faults is the program 1 in line numbers 11,16,22,30,32,40,44,47,48 which are given fault IDs as F1,F2,F3,F4,F5,F6,F7,F8,F9 respectively.

Table 5: Fault table for Program

Fault ID	Fault detected by			
	TC1	TC2	TC3	TC4
F1	*	-	*	*
F2	-	*	-	-
F3	-	-	*	-
F4	*	-	-	*
F5	*	-	-	-
F6	*	-	-	-
F7	*	-	-	-
F8	*	-	-	-
F9	*	-	-	-

By using the formula 1 the APFD for all approaches of test case prioritization were calculated as given below.

APFD = 1 – (TF1 + TF2 +TF3 +TF4+.....+ TFm) /(n*m) + 1/(2*n) ----------(1)

Where n is the number of faults and m is the number of test cases.

Let random (non-prioritized) test suite is { TC4,TC3,TC2,TC1} . APFD metrics for random (non-prioritized) test suite :

$$APFD = 1 - \frac{1+3+2+1+4+4+4+4+4}{4*9} + \frac{1}{2*4}$$

$$= 0.375$$

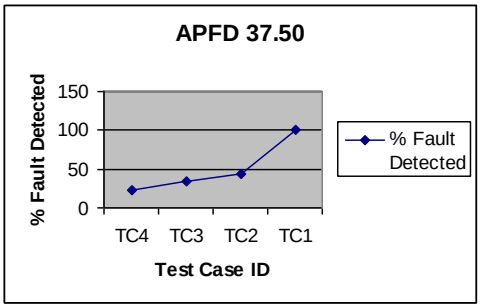

Fig 3 :APFD graph for non prioritized test suite of program

Table 6 : Different types of paths and lines of code covered by test cases in Prog

Test case ID	Total number of du-dc paths covered	Total number of non-dc paths covered	Total lines of code covered
TC1	18	06	42
TC2	08	04	19
TC3	08	04	24
TC4	07	04	31

Following the proposed approach to prioritize test cases, the prioritized test suite is as shown below : **{ TC1,TC3,TC2,TC4}**

APFD metrics for this prioritized test suite :

$$APFD = 1 - \frac{1+3+2+1+1+1+1+1+1}{4*9} + \frac{1}{2*4}$$

$$=0.79166$$

From the above calculations it is clear that prioritized test suite gives better APFD value. This is also shown in Fig 4.3 by APFD graph of prioritized test suite of program

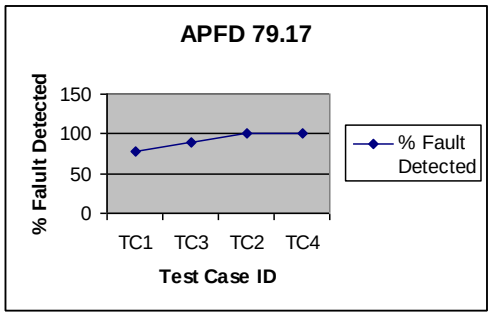

Fig 4: APFD graph for prioritized test suite of program

V. CONCLUSION AND FUTURE WORK

Data flow testing identifies potential bugs by examining the patterns in which that piece of data is used. But the problem with this testing is that there are large numbers of test cases thereby increasing the size of test suite. It may not be possible to execute all the test cases as it increases the time, effort and cost of the project. Therefore there is need to prioritize the test cases in data flow testing. A method has been proposed in this work in this direction. The proposed technique for test case prioritization of data flow test cases is based on non-dc-paths in the program. It is a beneficial technique for the purpose of saving resources such as time and cost. It proved to be effective because it identifies and executes the important test cases first and help in reducing the time, effort and cost of testing.

This study also has several limitations. These limitations can be addressed only through further studies of additional artifacts and regression testing techniques. The main limitation is that there is no approach to prioritize the test cases when the two test cases have the same number of lines of code.

REFERENCES

[1] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705{716, September 1995.

[2] H. Agarwal, J.R. Horgan, E.W. Krauser, S. London, "Incremental regression testing", IEEE International Conference on Software Maintenance, pp. 348-357, 1993.

[3] Maruan Khoury, Cost-Effective Regression Testing, 2006.

[4] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529-551, August 1996.

[5] S. Elbaum, A. Malishevsky, and G.Rothermel Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, February 2002.

[6] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, Alexey G. Malishevsky, Selecting a Cost-Effective Test Case Prioritization Technique, April 20, 2004.
[7] K.K. Aggarwal, Software Engineering, edition 3, New Age International publisher
[8] Dr. Naresh Chauhan ,"Software Testing – Principle and Practice" ,Oxford university press, 2010
[9] M.J. Rummel, G.M. Kapfhammer and A. Thall, " Towards the Prioritization of Regression test Suites with Data flow information", ACM Symposium on Applied Computing 2005.
[10] Y.Singh, A.Kaur and B.Suri, "Regression Test Selection and prioritization Using Variables-And Experimentation", Software Quality Professional, 2009,vol. 11, no. 2,pp. 38-51.
[11] Y.Singh, A.Kaur and B.Suri, "Emperical Validation of Variables based Test Case Prioritization/ Selection Technique", International Journal of Digital Technology and its Applications,vol3, no.3,September 2009.
[12] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodologies*, vol. 6, no. 2, pp. 173–210, Apr. 1997.
[13] S. Yoo and M. Harman, " Regression testing minimisation, selection and prioritisation : A survey," *Software Testing, Verification, and Reliability*, Mar. 2010.
[14] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, Sep. 2010.
[15] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

