

Finding the Effective Fault Localization and Test Case Prioritization Using Multiple Coverage Types

Mr. Mohan Raj¹ Mr. A. Kulothangan² Mr. Ajay Kumar³ Uzma Farooq⁴ Akthar Nazir⁵

^{1,2,3}Assistant Professor ^{4,5}M.Tech Student

^{1,2,3,4,5}Department of Computer Science and Engineering

^{1,2,3,4,5}SRM University

Abstract— To locate bugs in program is most tedious and time consuming activity which requires great effort on part of programmer, to locate exact location of faults is called as fault localization. There is a need to make fault localization process effective in its own unique and creative way for which work is already going on and many techniques exist for same e.g. branch, statement, data dependency, these techniques have their own advantages but also have few shortcomings like overhead of statements etc. our proposed technique focuses on fault localization and test case prioritization. By doing this our technique identifies fault as earlier as possible. Finally, our result analysis shows that our prioritization is more suitable for statement coverage, branch coverage and DU-pair coverage.

Key words: Fault localization, multiple coverage

I. INTRODUCTION

Debugging is one of the most expensive and time consuming processes for software developers. To address this expense, researchers have presented techniques to provide automated assistance in finding the faults that cause executions to produce incorrect outputs. Many of these techniques monitor runtime events to find those events that correspond to executions that fail. In particular researchers have investigated using the runtime coverage of entities such as statements and branches which require lightweight instrumentation and information flows, which require more expensive instrumentation. These researchers have shown empirically that techniques that use this coverage information provide guidance that can reduce the developers effort in searching for a faulty parts of the software There are many types of coverage information that fault localization techniques can utilize. However, to date there has been little research into which type of coverage maximizes fault-localization effectiveness or which type of lightweight coverage is best for fault localization in practical scenarios, such as deployed software, where monitoring overhead must be low. Our experiment explores the relative benefits of three lightweight types of runtime coverage monitoring—statements, branches, and du-pairs [1]—for use in the Tarantula fault-localization technique. Tarantula runs a test suite on the target program, assigns suspiciousness scores to statements, and ranks the statements from most suspicious to least suspicious. To perform a quantitative comparison of all three coverage types, and to provide an understandable view of branch and du-pair suspiciousness, we created a method that measures the effectiveness of branches and du-pairs in terms of statements.

We propose to use some parameters from all the three techniques via Statement Coverage, Branch Coverage and DU-pair simultaneously so in order to reduce cost that is required to localize faults and test case prioritization. By

doing this our technique our technique of finding the fault is as early as possible.

A. Software Testing:

Software testing is a process of executing a program or application with the intent of finding software bugs. It can also be stated as the process of validating and verifying that a software programme or application or product meets the business and technical requirements that guided its design and development. Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test, software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand risks of software implementation, it involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate extent to which the component or system under test. The various levels of testing are as follows:

1) Unit Testing:

During this first round of testing, the program is submitted to assessments that focus on specific units or components of the software to determine whether each one is fully functional. The main aim of this endeavor is to determine whether the application functions as designed. In this phase, a unit can refer to a function, individual program or even a procedure, and a White-box Testing method is usually used to get the job done. One of the biggest benefits of this testing phase is that it can be run every time a piece of code is changed, allowing issues to be resolved as quickly as possible. It's quite common for software developers to perform unit tests before delivering software to testers for formal testing.

2) Integration Testing:

Integration testing allows individuals the opportunity to combine all of the units within a program and test them as a group. This testing level is designed to find interface defects between the modules/functions. This is particularly beneficial because it determines how efficiently the units are running together. Keep in mind that no matter how efficiently each unit is running, if they aren't properly integrated, it will affect the functionality of the software program. In order to run these types of tests, individuals can make use of various testing methods, but the specific method that will be used to get the job done will depend greatly on the way in which the units are defined.

3) System Testing:

System testing is the first level in which the complete application is tested as a whole. The goal at this level is to evaluate whether the system has complied with all of the outlined requirements and to see that it meets Quality Standards. System testing is undertaken by independent testers who haven't played a role in developing the program. This testing is performed in an environment that closely mirrors production. System Testing is very important because

it verifies that the application meets the technical, functional, and business requirements that were set by the customer.

4) *Acceptance Testing:*

The final level, Acceptance testing (or User Acceptance Testing), is conducted to determine whether the system is ready for release. During the Software development life cycle, requirements changes can sometimes be misinterpreted in a fashion that does not meet the intended needs of the users. During this final phase, the user will test the system to find out whether the application meets their business' needs. Once this process has been completed and the software has passed, the program will then be delivered to production.

B. *Fault Localization:*

Finding and fixing bugs are essential and time consuming activities in software development. Once a bug is submitted, developers must allocate some effort to identify the exact location of the bug in source code. The problem of localizing bugs in a program is known as fault localization. It is a task in software debugging to identify the set of statements in a program, and the more developers participate in debugging. It is the activity of identifying the exact location of program faults; It is very expensive and time consuming process. It determines the root cause of failures, and also finds the cause of abnormal behaviour of fault program; it identifies exactly where bugs are located.

C. *Statement Coverage:*

The statement coverage is also known as line coverage or segment coverage, statement coverage covers only the true condition, through it we can identify the statements executed and where the code is not executed because of blockage, in this process we need to check each and every line of code and that needs to be executed.

Its advantages are that it verifies what a written code is expected to do and not to do, it measures the quality of code written, and it can also test whether the paths are tested or not. But it can also carry several disadvantages that it cannot test false conditions, and it also does not report whether loop reaches its termination condition, and it also cannot understand logical operators. It helps in ensuring that all the statements execute without any side effect, this method of testing comes under the category of white box testing.

D. *Branch Coverage:*

Branch coverage is also known as decision coverage or all edges coverage, it covers both true and false conditions unlike that of statement coverage, branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested

A decision is an if statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement. With an IF statement the exit can be TRUE or FALSE, depending on the value of logical condition that comes after IF. It is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

Its advantages are that it can validate that all the branches in code are reached; it also ensures that no branches lead to any abnormality of program's operation, it eliminates the problem that arises with statement coverage testing, it also

has few disadvantages that is this metric ignores branches within Boolean expressions which occur due to short circuit operators.

E. *Du- Pair Coverage:*

DU pair is a pair of definition and use for some variable, such that at least one DU path exists from the definition to the use

X = is a definition of x
= ...X is a use of x

A DU pair for variable x is a pair of nodes (n1, n2) such that

X is an DEF (n1)
The definition of x at n1 reaches n2
X is in USE (n2)

DU path a definition- clear path on the CFG starting from a definition to a use of same variable, their also exists some adequacy criteria for all DU pairs that is each DU pair is executed by at least one test case, for all DU paths each simple non looping DU path is executed by at least one test case.

II. PROBLEM DESCRIPTION

A. *Existing Problem:*

There exists many techniques to localize faults but based on our literature survey we take following three techniques into consideration:

- 1) Statement coverage based technique
- 2) Branch coverage based technique
- 3) DU-pairs coverage based technique

The above technique is used for only fault localization but our research includes consider finding the fault as early as possible and not given common technique to find the fault localization in all the above three types of coverage.

B. *Proposed Solution:*

Our research takes three existing techniques into consideration, our goal is to overcome the disadvantage of more cost involved to localize faults for which rather than mapping individually we map them simultaneously and hence reduce overall cost involved for fault localization. Finally ordering test cases that makes it efficient fault localization technique and finds fault as early as possible.

III. BLOCK DIAGRAM

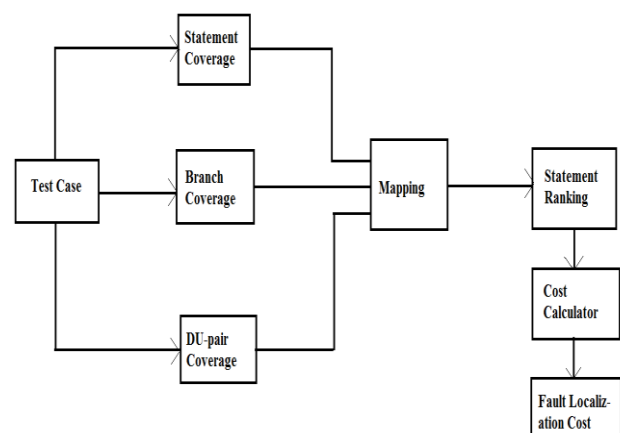


Fig. 1: Overall Block Diagram.

We have taken test case from test suite, that we feed as input to statement, coverage, branch coverage and Du- pair

and in existing technique it used to take all of them into consideration but individually, our goal is to take them simultaneously and map them so that we can perform ranking of statements as per their suspiciousness and then we calculate cost required to localize faults, but we reduce cost to localize faults as we map all of the techniques together.

IV. METHODOLOGY

ALGORITHM:

1. Test cases $T_j = \{T_1, T_2, T_3, \dots, T_n\}$
2. Statement $S_i = \{S_1, S_2, S_3, \dots, S_n\}$
3. for every test cases in T_j
4. for every statement in S_i
5. if T_j cover S_i
6. $C_{ij}=1$
7. else
8. $C_{ij}=0$
9. End if
10. End for loop
11. End for loop
12. Find total expected result(tt) and total no of false(tf) result
13. for each expected result
14. for each S_i
15. if $C_{ij}=1$;
16. $rt=rt+1$ //rt=row true
17. else
18. $rf=rf+1$;
19. End if
20. End loop
21. End loop
22. find the suspiciousness value of each S_i , $spe[i]=rf/(\sqrt{tf*(rf+rt)})$;
23. Print the C_{ij} and $spe[i]$;
24. Divide the branch statement of algorithm for true and false
25. for each branch statement
26. Repeat the statement from 3 to 23.
27. End for loop
28. Make the DU-Pairs from algorithm.
29. for each DU pair
30. Repeat the statement from 3 to 23.
31. End for loop;
32. End algorithm.

A. Algorithm 1:

Description of Algorithm 1, in this we have taken test cases like T1, T2.... From set T_j and we pass them to various statements like S1, S2...from set S_i , if test case is able to cover statement we say test case is executed and is marked as '1' otherwise we mark it as '0', based on the input of test cases we take some expected value and system may produce its own specified value which may not match with our expected value and this leads to fault case, then we need to calculate for suspiciousness which we calculate based on test

cases executed and not executed and those that are not executed.

V. CASE STUDY

A. Source Code:

```
Mid () {
  int x, y, z, m;
  1) read (x, y, z);
  2) m = z;
  3) if (y<z)
  4) if (x<y)
  5) m = y;
  6) else if (x<z)
  7) m = x;
  8) else
  9) if (x>y)
  10) m = y;
  11) else if (x>z)
  12) m = x;
  13) print (m);
}
```

Example 1

B. Statement Coverage:

It checks for true conditions, it ensures which statement is executed and which not, in this we have taken six test cases via T1, T2,.....,T6 and then on passing the values we give some expected value and system generates some value due to which their arises some fault case, in this case where in test case is executed is marked as '1' and test case not executed is marked as '0' and then we calculate suspiciousness based on test cases executed and not executed.

Suspiciousness is the mechanism of finding the cause that makes test case to fail or pass

$$\text{Suspiciousness (s)} = \frac{\text{failed (s)}}{\sqrt{\text{total failed} * (\text{failed (s)} + \text{passed (s)})}}$$

Where total failed is the total number of failing test cases, failed(s) is the number of failing test cases covering s and passed (s) is the number of passing test cases covering s.

Statem ents	T1	T2	T3	T4	T5	T6	Suspiciou sness
1	3,3 ,5	1,2 ,3	3,2 ,1	5,5 ,5	5,3 ,4	2,1 ,3	0.41
2	1	1	1	1	1	1	0.41
3	1	1	1	1	1	1	0.41
4	1	1	0	0	1	1	0.50
5	0	1	0	0	0	0	0.00
6	1	0	1	1	1	1	0.45
7	1	0	0	0	0	1	0.71
8	0	0	1	0	0	0	0.00

9	0	0	0	0	1	0	0.00
10	0	0	0	0	0	0	0.00
11	0	0	0	0	0	0	0.00
12	1	1	1	1	1	1	0.41
Pass/fail status	1	1	1	1	1	0	

Table 1: Statement Coverage

C. Branch Coverage:

It is also called decision coverage and it checks for both true and false conditions unlike that of statement coverage, in this case we take set of test cases viz T1, T2,.....,T6 and pass values to these test cases and it indicates '1' for test cases executed and '0' for not executed test cases.

Branches	T1 3,3 ,5	T2 1,2 ,3	T3 3,2 ,1	T4 5,5 ,5	T5 5,3 ,4	T6 2,1 ,3	suspiciousness
Entry	1	1	1	1	1	1	0.41
3 True	0	0	0	0	0	0	0.00
3 False	1	1	0	0	1	1	0.50
4 True	0	0	1	1	0	0	0.00
4 False	1	0	0	0	0	1	0.71
6 True	0	1	1	1	1	0	0.00
6 False	0	0	1	0	0	0	0.00
9 True	1	1	0	1	1	1	0.45
9 False	0	0	0	0	0	0	0.00
11 False	1	1	1	1	1	1	0.41
Pass/fail status	1	1	1	1	1	0	

Table 2: Branch Coverage

D. Du-Pair Coverage:

DU stands for definitions used, it is runtime coverage of program elements, in this the variables used in a program can be used elsewhere as well and in this we do pairing of statements and variables, and then we calculate suspiciousness.

DU-pairs	T1 3,3 ,5	T2 1,2 ,3	T3 3,2 ,1	T4 5,5 ,5	T5 5,3 ,4	T6 2,1 ,3	suspiciousness
1, 2, z	1	1	1	1	1	1	0.41
1, 3, y	1	1	1	1	1	1	0.41
1, 3, z	1	1	1	1	1	1	0.41
1, 4, x	1	1	0	0	1	1	0.50
1, 4, y	1	1	0	0	1	1	0.50
1, 5, y	0	0	0	0	0	0	0.00
1, 6, x	1	0	0	0	1	1	0.58
1, 6, z	1	0	0	0	1	1	0.58
1, 7, x	1	0	0	0	0	1	0.71
1, 9, x	0	0	1	1	0	0	0.00
1, 9, y	0	0	1	1	0	0	0.00
1, 10, y	0	0	1	0	0	0	0.00
1, 11, x	0	0	0	1	0	0	0.00
1, 11, z	0	0	0	1	0	0	0.00
1, 12, x	0	0	0	0	0	0	0.00
2, 13, m	0	0	0	1	1	0	0.00
5, 13, m	0	1	0	0	0	0	0.00
7, 13, m	1	0	0	0	0	1	0.71
10, 13, m	0	0	1	0	0	0	0.00
12, 13, m	0	0	0	0	0	0	0.00
Pass/fail status	1	1	1	1	1	0	

Table 3: DU- pairs coverage

1, 2, z	1	1	1	1	1	1	0.41
1, 3, y	1	1	1	1	1	1	0.41
1, 3, z	1	1	1	1	1	1	0.41
1, 4, x	1	1	0	0	1	1	0.50
1, 4, y	1	1	0	0	1	1	0.50
1, 5, y	0	0	0	0	0	0	0.00
1, 6, x	1	0	0	0	1	1	0.58
1, 6, z	1	0	0	0	1	1	0.58
1, 7, x	1	0	0	0	0	1	0.71
1, 9, x	0	0	1	1	0	0	0.00
1, 9, y	0	0	1	1	0	0	0.00
1, 10, y	0	0	1	0	0	0	0.00
1, 11, x	0	0	0	1	0	0	0.00
1, 11, z	0	0	0	1	0	0	0.00
1, 12, x	0	0	0	0	0	0	0.00
2, 13, m	0	0	0	1	1	0	0.00
5, 13, m	0	1	0	0	0	0	0.00
7, 13, m	1	0	0	0	0	1	0.71
10, 13, m	0	0	1	0	0	0	0.00
12, 13, m	0	0	0	0	0	0	0.00
Pass/fail status	1	1	1	1	1	0	

VI. CASE STUDY: MAPPING AND TEST CASE ORDERING

In this case study, we have taken three techniques into consideration via Statement Coverage, Branch Coverage and DU-pair Coverage and our main goal was to map them simultaneously so that the cost required to localize faults is reduced compared when we map them on individual basis.

For every technique like Statement coverage, branch coverage and DU-pair coverage, we have taken the suspiciousness which has highest value and accordingly checked for test cases that get executed at this value, which in every cases gets executed at T1, T6, in order to look for remaining cases we assigned weightage to each statement on random basis and then calculated values by adding that weightage and suspiciousness value together wherever test cases got executed and as a result of which initial ordering is of the form as follows:

For Statement Coverage:
 $T1 > T6 > T5 > T4 > T3 > T2$

For Branch Coverage:
 $T1 > T6 > T5 > T2 > T4 > T3$

For Du-pair Coverage:
 $T1 > T6 > T5 > T2 > T3 > T4$

But our goal is to map them simultaneously so that overall cost to localize fault is reduced which we have done by assigning values to test cases and then taking their average sum and hence it becomes of the form

$T1 > T6 > T51 > T42 > T33 > T24$
 $T1 > T6 > T51 > T22 > T43 > T34$
 $T1 > T6 > T51 > T22 > T33 > T44$

Now we will calculate its average sum e.g. for T5 it occurs at place 1 in all cases so its average will be taken as $1 + 1 + 1 = 3$ so $T5 = 3$, then for T4 it becomes $2 + 3 + 4 = 9$ and accordingly we get $T4 = 9$, following same procedure for other test cases we get some values through which we finally map them and hence it becomes of the form as follows:

$T1 > T6 > T3 > T4 > T2 > T5$.

VII. RESULT ANALYSIS:

In our result analysis we have analysed our subject program where in we plotted a graph between numbers of test cases vs. statements covered by particular test cases. We have plotted graph first for every technique and the mapped them into one which was our main goal.

A. For Statement Coverage:

The graph below is plotted between number of statements and test cases, where in we plot graph with respect to number of statements executed.

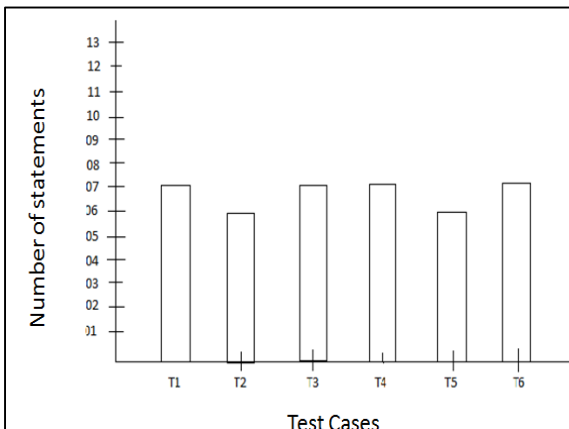


Fig. 2: Showing graph between number of statements and test cases for statement coverage technique.

B. For Branch Coverage:

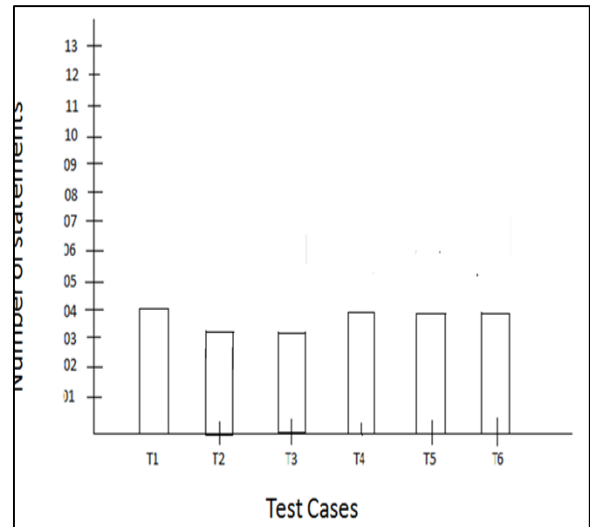


Fig. 3: Showing graph between number of statements and test cases for branch coverage technique.

C. For Du-Pair Coverage:

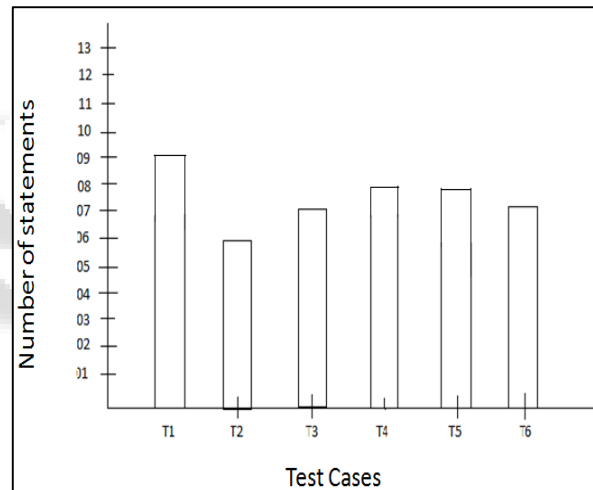


Fig. 4: Showing graph between number of statements and test cases for Du-pair coverage technique.

D. Test Case Ordering for Prioritization of Test Cases:

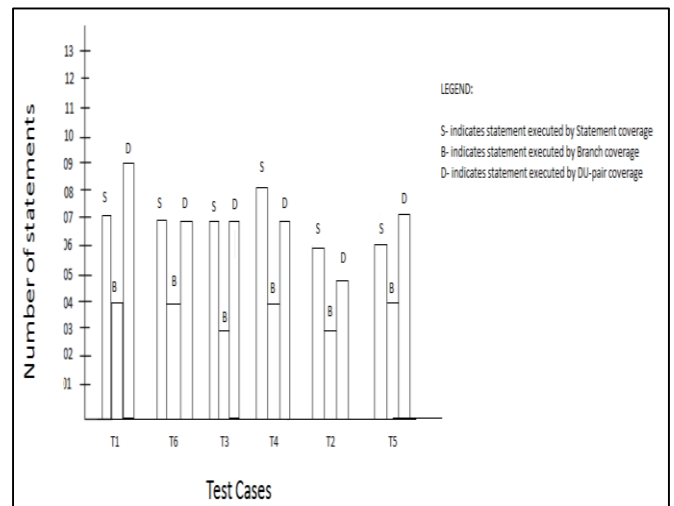


Fig. 5: graph plotted between number of statements and test cases where in we perform mapping and set the prioritization order of test cases.

VIII. CONCLUSION

In this paper we have taken three techniques into consideration viz statement coverage, branch coverage and du-pair coverage, which in the existing paper were taken separately and then mapped. But we have taken all of them together and mapped them simultaneously, finally this research shows that the approach proposed is best approach for fault localization as we find the fault as early as possible.

ACKNOWLEDGEMENT

Mr. Mohan Raj who had done his Bachelor of engineering from Anna University, Chennai and Master of Technology from shobhit university, Meerut. He is currently working as Assistant Professor in Department of Computer science engineering, SRM University. His area of interest includes software engineering, software testing, network security.

Miss Uzma Farooq who had done her Bachelor of Technology from BGSB University, Jammu and is currently pursuing Master Of Technology at SRM University, her area of interest include software testing, cryptography, graph theory.

REFERENCES

- [1] R. Abreu, P. Zoetewij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In Proc. Of TAIC-PART '07, pp. 89–98, Sep. 2007.
- [2] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In Proc. of Int'l Symp. on Softw. Reliability Eng., pp. 143–151, Oct. 1995.
- [3] G. K. Baah, A. Podgursky, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In Proc. of Int'l Symp. On Softw. Testing and Analysis, pp. 189–200, July 2008.
- [4] T. Ball and J. R. Larus. Efficient path profiling. In Proc. of 29th Int'l Symp. On Microarchitecture, pp. 46–57, Dec. 1996.
- [5] H. Cleve and A. Zeller. Locating causes of program failures. In Proc. of Int'l Conf. on Softw. Eng., pp. 342–351, May 2005.
- [6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In Proc. of European Conf. on Object- Oriented Programming, pp. 528–550, July 2005.
- [7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control flow based test adequacy criteria. In Proc. of Int'l Conf. on Softw.Eng., pp. 191–200, May 1994.
- [8] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In Proc. of Int'l Conf. on Automated Softw. Eng., pp. 273–282, Nov.2005.
- [9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In Proc. of Int'l Conf. on Softw. Eng., pp. 467–477, May 2002.
- [10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In Proc. of Conf. on Progr. Lang. Design and Impl., June 2005.
- [11] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In Proc. of European Softw. Eng. Conf. and Foundations of Softw. Eng., pp. 286–295, Sep. 2005.
- [12] W. Masri. Fault localization based on information flow coverage Technical report: AUB-CMPS-07-10. American University of Beirut, Computer Science, Oct. 2007.
- [13] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In Proc. of Int'l Conf. on Softw. Eng., pp.156–165, May 2005.
- [14] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In Proc. of Int'l Symp. on Softw. Testing and Analysis, pp.65–69, July 2002.
- [15] E. Renieris and S. Reiss. Fault localization with nearest neighbour queries. In Proc. of Int'l Conf. on Automated Softw.Eng., pp. 30–39, Oct. 2003.
- [16] R. Santelices and M. J. Harrold. Efficiently monitoring dataflow test coverage. In Proc. of Int'l Conf. on Automated Softw. Eng., pp. 343–352, Nov. 2007.