Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 10 Issue 05, May-2021

Fine Grained Statistical Debugging for the Identification of Multiple Bugs

Pruthviraj P
Electrical And Electronics Enginerring
PES Univesity
Bengaluru

***Abstract*:- Commercial software ships with undetected bugs despite the combined efforts of programmers, sophisticated bug detection tools and extensive testing. So, the identification and localization of the bugs in the software becomes essential issues in program debugging. Traditional software debugging is a difficult task to accomplish which requires a lot of time, effort and very good understanding of the source code. Given the scale and complexity of the job, automating the process of program debugging is very essential. Our approach aims at automating the process of program debugging. The earlier proposed approaches namely, statistical debugging and decision tree were able to identify only the most frequently occurring bugs and they failed to identify masked, simultaneously and non-frequently occurring bugs. We propose two approaches: one is decision tree based and the other uses bi-clustering for the task. The results obtained by our proposed approaches showed great improvements in the results in terms of purity, mis-classification rate and over splitting. Our proposed approaches were able to identify all the bugs present in the software including the masked and non-frequently occurring bugs.**

Keywords— Predicates, Predictors, Instrumentation, Purity, Mis-classification Rate, Oversplitting, Faulty Control Paths, Bug Localization, Bug Correction Introduction

1. INTRODUCTION

In a software cycle, more than 50% of the total cost may be spent on testing and debugging phases to ensure the quality of the software. So, developing effective and efficient debugging modules has become one of the important research topics. In standard terminology, program errors are defined as inappropriate actions committed by a programmer or a designer and bugs are the manifestations and results of errors during coding of the program. A program failure occurs when an unexpected result is obtained while executing the program due to the presence of bugs. In our setting, for the purpose of debugging, the programs are instrumented with predicates. A predicate is a boolean-valued expression over program variables. It can be a simple predicate or a compound predicate which is a boolean combination of simple predicates. We have used biclustering technique. The biclustering, co-clustering, or two-mode clustering is a data mining technique which allows simultaneous clustering of the rows and columns of a matrix. Two major steps involved in program debugging process are: localizing and correcting bugs in the software.

1.1 BUG LOCALIZATION AND CORRECTION

Bug localization is a step towards automated debugging. Code which is not related with the bugs is filtered out and the remaining code is checked for the presence of bugs. Consider a huge software like Rhythmbox which is a graphical open source music player on Linux operating system which has approximately 60,000 lines of code. In such a huge software, if some error occurs, manual debugging becomes a tedious and very difficult job. It requires a lot of programmer's time, effort and very good understanding of the source code. Effective bug localization techniques will save much of developers time and also they give insight on what caused the bug. In our setting, the bug localization module correctly finds bug predicting predicates. Then, the bug correcting module investigates the source code of the software to return the set of all faulty paths covered by these set of predicates. The set of faulty paths are then investigated to find out fault inducing transitions in the program and appropriate changes are made in the source code of the program to eliminate all of the bugs. The final bug correction is a manual process. The process of localizing the bugs is the most difficult among the two which needs to be automated. The focus of our work is to develop methods that automatically localize the bugs and thus help in debugging..

1.2 Overview of the Framework

Figure 1.1 shows the framework which we have used for program debugging. The debugging framework has several phases which are explained below. Instrumentation: This phase starts with a source-to-source transformation of a given buggy program. This transformation creates a lot of instrumentation sites in the program and instruments the program with predicates to collect data about the truth values of these predicates at particular program points. There are five categories of predicates that are looked for Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.

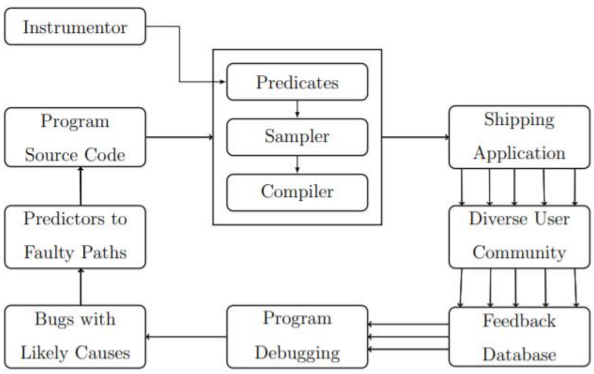

Figure 1.1: Program Debugging Framework

- Branches: For every conditional statement two predicates indicating whether true or false branch was taken are tracked.
- Loops: The predicate information related to while(condition), for(; ;), do{}while(condition) loops is collected.
- Returns: At each scalar returning function call site, six predicates are tracked indicating whether the return value is: 0, ≤ 0,=,≠,≥ 0 or >0
- Scalar Pairs: At each assignment statement x=< some_value>, identify each same-type in- scope variable y_i and each constant expression c_j . For each y_i and c_j , six predicates on the new value of x: <, ≤ 0,=,≠,≥ 0 or >0 are tracked. Each (x, y_i) and (x, c_j) pair is treated as a new instrumentation site i.e. a single assignment statement is associated with multiple distinct instrumentation sites.
- Pointer Null Test: Has two predicates and checks whether a pointer variable p_var==NULL or p_var≠NULL.

During run time the truth values of these predicates are recorded. A feedback report is generated for each run of the program. Each feedback report is a binary vector with a bit reserved for each predicate indicating whether the predicate was observed or not. Each feedback report also has a bit indicating whether the run was successful or failed. Since the predicates instrumented are automatically generated, most of them do not provide any useful information regarding the bugs. So, the primary task is to select the most useful bug predicting predicates from the set of instrumented predicates.

Shipping: The application is shipped to the end users who will then install the software on their systems based on the requirements. Feedback reports are collected and sent from the end users to the developers for both successful runs as well as for the failed runs. The feedback reports sent from the end users are stored in feedback database.

Program Debugging: This is the heart of whole program debugging framework. It makes use of the feedback reports collected from thousands of users and uses data mining and statistical techniques to identify and localize the bugs in the software. Finally, it returns a set of predicates which are directly associated with bugs.

Bug Localization: Program debugging phase will return a set of predicates which correctly identify the bugs. If the source code is huge, then the number of predicates instrumented will be high as well and the set of predicates returned by the program debugging phase will also be high. This again requires a lot of manual work to actually localize bugs. The bug localization phase uses data mining techniques to give a set of faulty paths covered by these set of predicates containing fault inducing transitions in the program. This further reduces the amount of manual work needed for program debugging.

Source Modification: Once the set of faulty control paths are found, appropriate changes are made in the source code to eliminate all the bugs and the software is redistributed. Now, new predicates may also be instrumented in the source code for observing changed behavior of the program.

2. PROBLEM STATEMENT

In the presence of multiple and simultaneously occurring bugs, the most frequently occurring bugs will dominate and they will mask other non frequently occurring bugs. Recent work in the field of program debugging aims at finding out the most frequently occurring bugs. The earlier approaches, namely statistical debugging and decision tree fail to identify multiple, simultaneously and non-frequently occurring bugs. Our current work aims at finding out all the bugs present in the software including multiple, simultaneously and non-frequently occurring bugs. Our approach makes use of data mining and statistical techniques in achieving the goal.

We now introduce some terminology.

P - Set of instrumented predicates

R - Set of feedback reports where each report is a binary vector with a bit reserved for each predicate indicating whether the predicate was observed or not during the run.

S/F - Each feedback report has a bit indicating whether the run was successful or failed. The final outcome of the debugging algorithm is a subset of the powerset of failed runs. Table 2.1 gives a sample set of feedback reports. Here, rows represent feedback reports, columns represent predicates and the last column represents the exit status of the program.

We propose two algorithms: one is based on decision tree method and the other uses biclustering. The evaluation of the algorithms are done based on the following measures

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	S/F
R_1	0	1	0	0	1	1	0	1	F
R_2	0	1	0	1	0	0	1	0	F
R_3	0	0	0	0	0	0	0	0	S
R_4	0	1	0	1	1	1	0	0	S
R_5	1	1	0	0	1	0	0	0	F
R_6	0	1	0	0	0	1	0	1	F
R_7	0	0	0	0	0	0	1	0	S
R_8	0	1	0	1	0	1	0	0	F

Table 2.1: Debugging Information

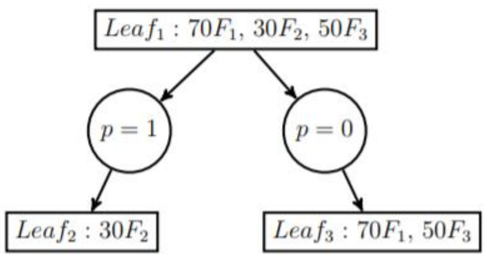

Figure 2.1: Purity Checking

Purity: Purity with respect to a set of failed runs tells whether the set has bugs from only one bug class. If the set has bugs from only one bug class, the set is treated as pure. Otherwise, it is treated as impure. In figure 2.1, $Leaf_2$ is pure as it has reports only from F_2, whereas $Leaf_3$ is impure as it has reports from two bug classes namely, F_1 and F_3.

Mis-classification Rate: Mis-classification rate with respect to a bug class gives the number of bug reports of the bug class wrongly classified as bug reports belonging to other bug classes. In figure 2.2, 70 reports from F_1 and 30 reports from F2 are split across $Leaf_2$ and $Leaf_3$. There is a mis-classification of 20 reports of F_1 and 10 reports of F_2. Overall, there is a mis-classification of 30 reports out of 100 i.e. the mis-classification rate is 30%.

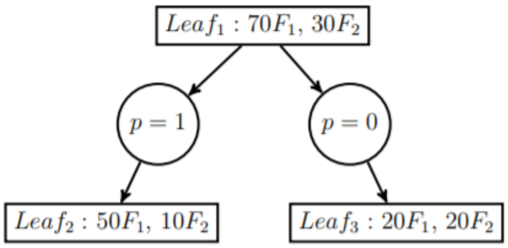

Figure 2.2: Mis-classification Calculation

Oversplitting: Oversplitting with respect to a bug class tells whether the bug reports of the bug class are distributed across several sets of failed runs or a single set. If the bug reports are distributed across two or more sets, then the bug class is said to be oversplit. Oversplitting measure (OM) is evaluated using the following simple formula.

$$OM = \sum_{x \in BugClass} (numSets_x - 1)$$

where, $numSets_x$ denotes the number of sets of failed runs containing bug reports from the bug class x.

If all the reports of the bug class are present in only one set, then its OM is zero which tells that the bug class is not oversplit. Otherwise, the bug class is said to be oversplit. In figure 2.2, each of bug classes F_1 and F_2 have an oversplitting measure of 1 which tells us that the bug classes have been oversplit, whereas in figure 2.1, each of the bug classes F_1, F_2, F_3 have an oversplitting measure of 0 which tells us that the bug classes are not oversplit.

2.3 Summary Of Results

We carried out our experiments on Nokia Siemens suite which is available at Software Infrastructure Repository (SIR) and compared our results with earlier approaches namely statistical debugging proposed by Liblit and decision tree methods. The results obtained from these two approaches clearly showed the need of postprocessing so as to increase the purity of partitions, reduce the rate of mis-classification of bugs and to avoid oversplitting.

The two proposed approaches: one based on decision tree method and the other using biclustering showed a great improvement in the results. There was a drastic reduction in the mis-classification rate and most of the bug partitions were pure. The results showed a slight oversplitting. As each bug can get triggered by different control flow paths in the program, bug reports from a particular bug class can get distributed across several sets of failed runs signifying that they were bug reports from a single bug class but possibly triggered by different conditions in the program. The predicates returned by our approach were able to localize all the bugs present in the sofware including masked and non-frequently occuring bugs which statistical debugging and decision tree failed to find out.

Table 2.2 summarizes the results obtained using statistical debugging and one of our proposed approach namely postprocessing using decision tree based method. Clearly, we can see that there is a drastic reduction in the mis-classification rate and most of the partitions are pure. Oversplitting measure signifies us that a bug can get triggered by different conditions in the program. So, its bug reports gets distributed across multiple bug partitions. SD in Table 2.2 represents statistical debugging technique and FPR represents one of our proposed approach for postprocessing, which is a decision tree based method. We have also conducted experiments using decision tree, postprocessing using biclustering. The detailed results of all the experiments are shown in results section.

Case Study	Over splitting		% of bugs mis-classified		% of pure leaves	
	SD	FPR	SD	FPR	SD	FPR
1	14	26	12	6	36	87
2	17	24	18	12	45	74
3	22	22	44	27	50	60
4	9	12	13	4	0	70
5	5	6	8	3	50	90
6	3	5	12	1	55	75
7	22	33	47	21	35	81

Table 2.2: SD Importance(P) vs Phase2 FPR

3. PROPOSED ALGORITHMS

3.1 Terminology

Input to the debugging framework consists of a set of instrumented predicates and feedback reports from the end users. The notations used in the description of our approach are summarized in Table 3.1.

Output from the debugging module is of the form:

$A = \{(P_1, F_1),(P_2, F_2), ...,(P_l , F_l)\}$

s.t $\forall i, j, i \neq j, P_i \cap P_j = \Phi$ and $F_i \cap F_j = \Phi$ and $\forall i, P_i \subseteq P$ and $F_i \subseteq R_F$. $\forall i, i \leq 1 \leq l$,

F_i represents the bug i and P_i represents the set of predicates which can identify and localize the bug i correctly. For every i, P_i represents a set of paths in the program's control flow graph where the bug F_i can be found.

Figure 3.1 represents the final output in a decision tree format.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 10 Issue 05, May-2021

3.2 Algorithms

Phase 1, Statistical Debugging:

Algorithm 1 is the statistical debugging technique used for program debugging. It tries to associate each failed run with a predicate, which can identify the particular failed run. The predicate is said to be the predictor of the failed run.

Notation	Meaning
P	Set of instrumented predicates.
R	Set of feedback reports or runs.
R_F	Set of failed runs, $R_F \subseteq R$.
R_S	Set of successful runs, $R_S \subseteq R$.
F_i	Set of failed runs in partition i, $F_i \subseteq R_F$.
S_i	Set of successful runs in partition i, $S_i \subseteq R_S$.
P_i	Set of predicates in partition i, $P_i \subseteq P$.
$n(R_F)$ or $NumF$	Total number of failed runs.
$n(R_S)$	Total number of successful runs.
$n(p, R_F)$ or $F(p)$	Total number of failed runs in which the predicate is true.
$n(p, R_S)$ or $S(p)$	Total number of successful runs in which the predicate is true.
$n(p \text{ observed}, R_F)$	Total number of failed runs in which the predicate was observed at all(either as true or false).
$n(p \text{ observed}, R_S)$	Total number of successful runs in which the predicate was observed at all(either as true or false).
$\varepsilon(R)$	Entropy of partition having failed runs set F and successful runs set S.
$\varepsilon(R_p)$	Entropy of partition having p to be true.

Table 3.1: Notations and their meaning

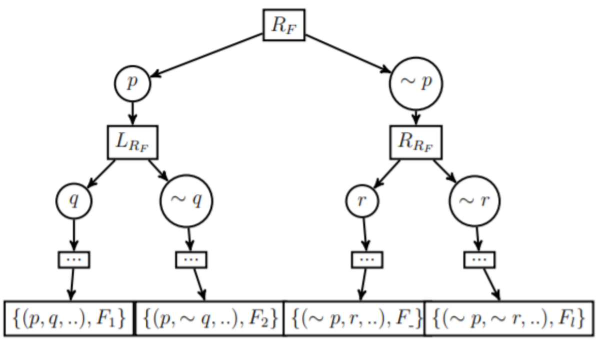

Figure 3.1: Final output in Decision tree format.

Statistical debugging uses 3 different measures to rank predicates. They are listed below. ∀p ∈ P, Rank(p)=

1. Failure(p) $= \frac{n(p,R_F)}{n(p,R_F)+n(p,R_S)}$

2. Increase(p)= Failure(p)-Context(p)=
$$\frac{n(p,R_F)}{n(p,R_F)+n(p,R_S)} - \frac{n(p\ observed,R_F)}{n(p\ observed,R_F)+n(p\ observed,R_S)}$$

3. Importance(P)=$\frac{2}{\frac{1}{Increase(p)}+\frac{1}{\frac{log(n(p,R_F))}{log(NumF)}}}$

The statistical debugging approach proposed by Liblit is a decision list based technique. A decision list is a list of nodes, where each node is formed based on the decision taken at the earlier node.

In algorithm 1, a simple iterative elimination approach is used. This forms the phase 1 of our proposed approach.

The steps of the algorithm are explained below.

1. The predicates are ranked based on Importance(p) measure. The predicates with

Algorithm 1: STAT_DEBUG( )

```
// Statistical Debugging using Importance(p)
repeat
    forall the p ∈ P do
        calculate Importance(p);
    end
    Rank set P based on Importance;
    top_pred = argmax⁺_{p∈P} {Importance(p)};
    All Runs in which top_pred is true form a leaf;
    top_pred is the bug indicator for the leaf;
    Mark this leaf for Phase 2: Phase_LEAF = 2;
    R = R\{Runs in which top_pred is true};
    P = P\{top_pred};
until Size(P) = 0 || Size(R_F) = 0;
return;
```

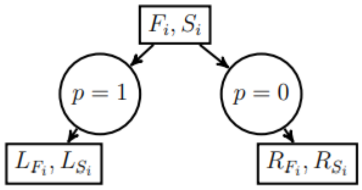

Figure 3.2: The way reports are split based on p.

Increase(p) ≤ 0 have no bug predicting power. So, all such predicates are eliminated. The top ranked predicate is then picked. Assume p as the top ranked predicate.

2. Split sets F_i and S_i (set of failed and successful runs in ith iteration of the algorithm), each into two sets L_{Fi} , R_{Fi} and L_{Si} , R_{Si} respectively based on the top ranked predicate p. L_{Fi} and L_{Si} are the failed and successful runs respectively having predicate p to be true. They form left child of the partition i. R_{Fi} and R_{Si} are the failed and successful runs respectively having predicate p to be false. They form right child of the partition i. The top ranked predicate p is the bug predictor for the partition F_i . Figure 3.2 shows how the set i is split based on predicate p.

3. Repeat steps 1-2 on right child of the partition i until the set of failed runs becomes empty or the set of predicates becomes empty.

4. All the leaves are marked for postprocessing during phase 2.

Phase 1, Decision Tree:

. It uses information gain measure to rank the predicates. The goal of the decision tree is to get the maximum utility. In our case, the utility is to get as many pure bug partitions as possible. The steps of the algorithm are explained below.

1. The predicates are ranked based on information gain measure. Assume p as the top ranked predicate.

2. This step is same as step 2 of Algorithm 1.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 10 Issue 05, May-2021

3. Repeat steps 1-2 on both left as well as the right child of partition i until the set of predicates becomes empty or the stopping criterion is met.

The above algorithms fail to capture multiple, simultaneously and non-frequently occurring bugs. They try to find only the most frequently occurring bugs. As already mentioned, in the presence of multiple and simultaneously occurring bugs, the most frequently occurring bugs will dominate and they will mask other non frequently occurring bugs. So, there is a need for postprocessing to find out multiple, simultaneously and non-frequently occurring as well as the masked bugs.

We propose two postprocessing approaches. They are described below.

Phase 2, Frequent Predicate Ranking:

Algorithm 2 is the first proposed approach for postprocessing. The following algorithm is a decision tree based technique which uses F(p) measure as the ranking function to capture all the masked and non-frequently occurring bugs

Algorithm 2: PHASE2_FPR(Leaf L)

```
// Postprocessing using Frequent Predicate Ranking.
if size(R_F) = 0 || size(P) = 0 || height ≥ Max_height then
    return;
forall the p ∈ P do
    calculate n(p, R_F);
end
Rank set P based on n(p, R_F);
top_pred = argmax^+_{p∈P} {n(p, R_F)};
Runs in which top_pred is true form the left child;
Runs in which top_pred is false form the right child;
top_pred is the bug indicator for the left child;
P = P\{top_pred};
// Recursive call on both left as well as the right child
PHASE2_FPR(Left Child);
PHASE2_FPR(Right Child);
return;
```

All the leaves marked for phase 2 in Algorithm 1 are processed here. The steps of the algorithm are explained below. For every leaf,

1. If the user's choice is to retain the predicates, to allow predicates used on the other subtree to be picked again as the top predicates in this subtree, then the procedure mark predicates is called. It marks the predicates on the path from leaf to root as used and all other predicates as unused. If the user's choice is to discard predicates, a predicate once used is permanently eliminated from the set of predicates.

2. The predicates are ranked based on F(p) measure. Assume p as the top ranked predicate.

3. Split partitions Fi and Si each into two partitions L_{Fi} , R_{Fi} and L_{Si} , R_{Si} respectively based on the top ranked predicate p. L_{Fi} and L_{Si} are the failed and successful runs respectively having predicate p to be true. They form left child of the partition i. R_{Fi} and R_{Si} are the failed and successful runs respectively having predicate p to be false. They form right child of the partition i. The top ranked predicate p is the bug predictor for the partition F_i . Both, the left as well as the right child of the partition i are marked for further postprocessing.

4. Repeat steps 1-2 on both left as well as the right child of partition i until the set of predicates becomes empty or the stopping criterion is met. In this approach height is used as the stopping criterion. The maximum height allowed is maximum height of statistical debugging approach plus (3 to 4 additional height). We chose 3 to 4 as the additional height because, the case study which we have used for conducting our experiments had multiple bugs, the number varying from 4 to 5 and three to four iterations of our proposed approach at each leaf node was able to identify the bugs.

5. Finally, at each leaf partition i we will obtain the form (P_i , F_i) using sets P_i and F_i , where P_i contains all the predicates on the path from partition i to the root and F_i contains the set of failed reports at the leaf partition i which usually are bug reports from a single bug class. Predicates in the set P_i identify and localize the bug F_i correctly.

Phase 2, Bi-clustering:

Algorithm 3 is the second proposed approach for postprocessing. It tries to associate the set of failed runs with the set of predicates which can identify those runs correctly. It employs biclustering technique to simultaneously cluster the set of failed runs and the set of predicates. The final output is a set of clusters where each of the cluster has bug reports from a single bug class and the cluster also has predicates identifying the reports of the bug class correctly

Algorithm 3: PHASE2_BC( )

```
// Postprocessing using Bi-clustering. Leaves contain only
   failed runs.
forall the leaf ∈ Leaves do
    {C1, C2}=BICLUST(leaf,num_Clusters=2);
    foreach Cluster do
        make p with max Increase as predictor;
    end
    R ∈ Leaf but ∉ {C1, C2} forms third partition;
end
return
```

All the leaves marked for phase 2 in Algorithm 1 are processed here. The steps of the algorithm are explained below. For every leaf,

1. Use biclustering to cluster the set of failed runs and the set of predicates simultaneously. It uses only F_i , the set of failed runs at leaf partition i and only those predicates which have Increase(p) > 0 and are not used anywhere before as the top ranked predicates. As the information regarding the number of bugs and their types is not available, assuming there can be atmost two bug classes at each of the leaf partition, the number of clusters is restricted to two.

2. The two clusters returned will form the left and right child of the partition i namely L_{Fi} and R_{Fi} . The set of failed reports in Fi which does not belong to any of the two newly formed clusters will form the partition W_{Fi} .

3. For each cluster, pick the predicate with maximum Increase value as the representative of the cluster.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 10 Issue 05, May-2021

4. Finally, at each leaf partition i we will obtain the form (P_i , F_i) using sets P_i and F_i , where P_i contains all the predicates on the path from partition i to the root and Fi contains the set of failed reports at the leaf partition i which usually are bug reports from a single bug class. Predicates in the set P_i identify and localize the bug F_i correctly.

Biclustering algorithm takes binary matrix M^{mn} and N, the number of clusters desired and returns N different clusters. The algorithm is described below.

1. Using first row as a template, first the set of columns of input matrix M i.e. the set of predicates P is divided into two subsets P_U and P_V . The predicates having value 1 in the first row are moved to set P_U and those predicates having the value 0 are moved to set P_V .

2. Now, the rows of matrix M i.e. set of runs R are rearranged: first come all the runs that have predicates only from set P_U , then those runs that have predicates from sets P_U and P_V , and finally the runs that have predicates from set P_V only. The corresponding sets of runs R_U , R_W and R_V in combination with P_U and P_V forms submatrices U and V

3. The idea behind the algorithm is to divide the input matrix into three matrices, one of which contain only 0-cells. So, it is discarded. The algorithm is then recursively applied to the remaining two submatrices U and V; the recursion ends if the current matrix represents a bicluster, i.e., contains only 1s. If U and V do not share any rows and columns of M, i.e., R_W is empty, the two matrices can be processed independently from each other. However, if U and V have a set R_W of rows in common, special care is necessary to only generate those clusters in V that share at least one common column with P_V .

4. The parameter Z serves this goal. It contains sets of columns that restrict the number of admissible clusters. A bicluster (R, P) is admissible, if (R, P) share one or more columns with each column set P + in Z, i.e., $\forall p^+ \in Z : P \cap P^+ \neq \Phi$.

5. Finally, the program returns N number of clusters

4. NUMERICAL EXPERIMENTS AND RESULTS

Table 4.1 provides the implementation details. The process of instrumentation and collecting feedback reports is fully automated. Statistical Debugging is the bug isolation project proposed by Liblit & Co. Decision Tree is the standard decision tree method which uses information gain measure for ranking. Frequent Predicate Ranking and Postprocessing using Bi-Clustering are our proposed approaches.

	Language	LOC
Automated Instrumentation	Java	300
Phase 1, Statistical Debugging	Java	800
Decision Tree	Library	500
Phase 2, Frequent Predicate Ranking	Java	1500
Phase 2, Postprocessing using Bi-Clustering	Java	1200

Table 4.1: Implementation Details

Case Study	Program	LOC	No. of Bugs	No. of Predicates	No. of bug Reports	Total no. of Tests
1	printtokens	964	4	186	210	4071
2	printtokens2	792	5	410	909	4071
3	replace	1155	5	150	705	5542
4	schedule	427	4	48	419	2650
5	schedule2	461	4	210	96	2710
6	tcas	197	5	32	309	1608
7	totinfo	560	5	146	365	1052

Table 4.2: Dataset

Case study	Height of tree	No. of leaves	Size of (P) used	Over splitting	% of bugs mis-classified	% of pure leaves
1	11	11	14	14	12	36
2	11	11	13	17	18	45
3	4	4	4	22	44	50
4	5	5	5	9	13	0
5	4	4	4	5	8	50
6	5	5	5	3	12	55
7	7	8	7	22	47	35

Table 4.3: Importance(P) output

Case study	Height of tree	No. of leaves	Size of (P) used	Over splitting	% of bugs mis-classified	% of pure leaves
1	5	9	8	21	11	33
2	8	29	19	29	4	62
3	14	25	21	72	34	16
4	14	25	18	26	6	64
5	4	7	6	11	5	67
6	3	4	3	7	13	75
7	8	19	12	27	23	40

Table 4.4: Decision Tree output

Case study	Height of tree	No. of leaves	Size of (P) used	Over splitting	% of bugs mis-classified	% of pure leaves
1	14	31	23	26	6	87
2	16	27	25	24	12	74
3	8	10	10	22	27	60
4	9	13	12	12	4	70
5	8	11	10	6	3	90
6	5	8	6	5	1	75
7	14	32	22	33	21	81

Table 4.5: Phase2 FPR output

Case study	Height of tree	No. of leaves	Size of (P) used	Over splitting	% of bugs mis-classified	% of pure leaves
1	12	22	33	22	2	81
2	13	22	35	24	16	77
3	5	8	12	25	31	100
4	6	11	15	11	3	54
5	5	7	12	7	3	71
6	6	6	12	5	9	66
7	12	20	32	36	29	65

Table 4.6: Phase2 BC output

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 10 Issue 05, May-2021

Table 4.7 summarizes the results obtained using all the four approaches

	Most frequent Bugs	Masked bugs	Purity	Over-splitting	Mis-Classification
SD	Yes	No	Less	Yes	More
DT	Yes (a few)	No	Less	Yes	More
FPR	Yes	Yes	More	Yes	Very Less
BC	Yes	Yes	More	Yes	Very Less

Table 4.7: Summary of Results

5. CONCLUSIONS

We have described a suite of instrumentation and analysis techniques for diagnosing bugs in widely deployed software. The earlier approaches namely, statistical debugging and decision tree failed to capture the masked and non-frequently occurring bugs. We have proposed two postprocessing approaches for the task: one is based on decision tree method and the other uses biclustering . The results obtained showed great improvements in the results in terms of purity, mis-classification rate and oversplitting. Our proposed approaches were able to identify all the bugs present in the software including the masked and non-frequently occurring bugs. Our proposed approaches uses decision tree and biclustering methods. For biclustering , methods like Non-negative matrix factorization can be thought as a good alternative.

6. REFERENCES

[1] ANSI/IEEE. IEEE Standard Glossary of Software Enginnering Terminology. IEEE Std 729-1983. IEEE, New York, 1983.
[2] Rhythmbox: http://developer.gnome.org/rhythmbox/.
[3] SIR: Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/ index.html.
[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In POPL: Principles of Programming Languages, pages 97-105, 2003.
[5] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In Machine Learning, Proceedings of the Twenty-Third Interna tional Conference (eds. W.W. Cohen and A. Moore) 161-168 (ACM, New York, 2003).
[6] Y. Cheng and G. M. Church. Biclustering of expression data. In Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00), pages 93-103, 2000.
[7] T. M. Chilimbi, A. V. Nori, and K. Vaswani. Quantifying the effectiveness of testing via efficient residual path profiling. In FSE: Foundations of Software En gineering, pages 545-548, 2007.
[8] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In Proceedings of the International Conference on Software Engineering (ICSE), May 2009.
[9] H. Cleve and A. Zeller. Locating causes of program failures. In ICSE: International Conference on Software Engineering, pages 342-351, 2005.
[10] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In ASE'08: Automated Software Engineering, pages 184-193, 2007.
[11] A. Klivans and R. Servedio. Toward attribute-efficient learning of decision lists and parities. In Seventeenth Annual Conference on Computational Learning Theory (COLT), 2004, pp. 234-248.
[12] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In Advances in Neural Information Processing Systems, volume 13, pages 556-562, 2001.
[13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In J. James B. Fenwick and C. Norris, editors, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03), volume 38, 5 of ACM SIGPLAN Notices, pages 141- 154. ACM Press, 2003.
[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistiscal bug isolation. In PLDI: Programming Language Design and Implementation, pages 15-26, 2005.
[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In OOPSLA 2010.