Iterative Path Clustering for Software Fault Localization

Rong Chen, Shifeng Chen
Dalian Maritime University
Dalian, China
rchen@dlmu.edu.cn

Nan Zhang
Shenyang Institute of Engineering
Shenyang, China
zn491@163.com

***Abstract*—A number of studies have been done to pinpoint program faults, among them the testing-based fault localization (TBFL) technique does with ranking suspiciousness of statements by counting statement occurrences in failing and successful runs of a buggy program. However, the downgrade of TBFL is its sensitiveness to the distribution of execution paths produced by test cases, so there still exists the need for improvement, given the great variety and uncertainty of execution paths. Path clustering can not only reduce redundancy, but also casts light on understanding test cases. This paper proposes a fuzzy *c*-Lines clustering for classifying execution paths and understanding faults iteratively and interactively. In doing so, we figure out vector angles between failed and passed paths to group them into categories, from which we elect some for statistical comparison via TBFL. Empirical results show that our method can help to improve the efficiency and accuracy in locating and understanding faults.**

Keywords- Testing Based Fault Localization; Fuzzy c-Lines Clustering; Algorithm; Path Descriptor

I. INTRODUCTION

As debugging is one of the most expensive and time-consuming processes for software development, computer-aided software debugging can help to reduce time to market and the overall costs for software development [1]. In the past decades, there has been a high demand for automatic fault localization that can guide programmers to the locations of faults with minimal human intervention [2,6].

There has been a body of work on diagnosing program bugs both in the field of software engineering and artificial intelligence (AI), including program slicing (PS) [5], Delta Debugging (DD) [6], TBFL [2-4,13,14], error explanation (EE) [15], specification assisted fault localization (SAFL) [16] and Model-Based Software Debugging (MBSD) [17]. While PS identifies a set of statements which affect variable values at certain program point, DD searches for faulty statements by alternating program states, and TBFL does with using the statistical comparison of coverage information in a number of execution paths. More formal than DD and TBFL, EE applies the Bounded Model Checking on C programs to find counterexamples and searches for nearest evidences by SAT solvers. SAFL pinpoints statements that lead to inconsistency with program specifications such as data structures properties, whereas MBSD adopts model-based diagnosis, as developed in AI for many years, to diagnose program errors by using a program model, which describes a working piece of software in terms of structure and behavior. If the software does not behave as expected, a diagnostic system will conclude which statement or expression accounts for this misbehavior.

So far considerable amount of effort has been made to enhance automated debugging, and a number of studies have confirmed that TBFL exhibits effectiveness in coping with real programs. But as will be seen in Section 2, there are also various cases where available techniques behave undesirably, in particular when there is no obvious difference in execution paths or a fault is executed but does not manifest itself persistently. In fact, most TBFL techniques have assumed that differences in the spectra (abstraction of execution paths) are signs of faults and they are persistently manifested. This is OK for almost-correct and inherently deterministic programs, but vulnerable when it comes to programs producing almost identical execution paths. Clustering methods, native to AI, have not been widely used in fault localization, except for identifying coincidental correctness [7] and mining sequential patterns [8].

The goal of this work is to reduce the adverse impact of test case distribution on TBFL's performance, by grouping 'similar' execution paths. By similarity we mean either execution paths are identical, or they have a very short distance in program state space, by measuring their angles in *c* hyper-lines. To assist fault localization, we calculate vector angles between all execution paths to partition them into several categories, from each, failed or passed group, we select one representative and apply existing techniques like TBFL to pinpoint the real culprit. The focus of this work is a supplement that supposedly enhances fault localization technique (e.g. Tarantula and Ochiai), or revs up better understanding of program errors by checking the difference between a series of path descriptors (i.e. a conjunction of assertions describing execution paths).

敏感
精确找到
退步
不一致的
易受攻击的
完全相同的
不利的影响
介入
诊断

978-1-5090-3713-1/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS-C.2016.85

The contributions of this paper are threefold: (1) we present fuzzy *c*-lines path clustering as a supplement to fault localization, (2) our approach features fault understanding by comparing path descriptors of failed and passed execution paths, queried iteratively and interactively, and (3) we show the diagnostic accuracy of our approach through experiments on the well-known Siemens benchmark, in comparison to earlier techniques.

II. A MOTIVATING EXAMPLE

Available techniques adversely degrade especially when execution paths are similar, as argued earlier. A disappointing example qr(x,y) is shown in the leftmost column of Table I. This program is expected to figure out the remainder and quotient of input variables x and y, but it contains a fault in line 4 which should be r>=y. Thirteen columns in the middle show execution paths produced by test cases t_i (for all i, $1 \le i \le 13$), 9 of them passed and 4 failed, respectively denoted by 'P_i' and 'F_i' at the bottom of each column (where i is the sequence number in passed or failed execution paths).

For every test case t_i, its values for input variables x and y are placed right below itself, and each column t_i depicts a program run (i.e. execution paths) with these input values, where the execution number of a statement is placed at the intersection of the statement row and the test case column. What is clear in Table 1 is that the execution path of t_2 and that of t_4 are the same, so are those of t_5 and t_6, and the same thing also happens to t_7 and t_8, to t_9, t_{10} and t_{12}, and to t_{11} and t_{13}, as well.

TABLE I. AN EXAMPLE OF FAULTY PROGRAM (LEFT); TEST CASES; RESULTS (MIDDLE) AND EXECUTION PATHS

Program	Test Cases													TBFL	
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	Ta	Oc
int qr(int x, int y){	3,3	6,3	9,3	4,2	1,3	2,3	4,3	5,3	7,3	8,3	10,3	5,2	11,3	-	-
1: int r, q;	1	1	1	1	1	1	1	1	1	1	1	1	1	-	-
2: r=x;	1	1	1	1	1	1	1	1	1	1	1	1	1	0.50	0.58
3: q=0;	1	1	1	1	1	1	1	1	1	1	1	1	1	0.50	0.58
4: while (r>y) { //should be r>=y	1	2	3	2	1	1	2	2	3	3	4	3	4	0.50	0.58
5: r=r-y;	0	1	2	1	0	0	1	1	2	2	3	2	3	0.67	0.71
6: q=q+1; }	0	1	2	1	0	0	1	1	2	2	3	2	3	0.67	0.71
7: return q;}	1	1	1	1	1	1	1	1	1	1	1	1	1	0.50	0.58
Pass/Fail Status	**F_1**	**F_2**	**F_3**	**F_4**	**P_1**	**P_2**	**P_3**	**P_4**	**P_5**	**P_6**	**P_7**	**P_8**	**P_9**	**-**	**-**

Even though utilizing the whole runtime information in 13 execution paths, TBFL technique like Tarantula (abbreviated as Ta) and Ochiai (abbreviated as Oc) give the highest suspicious score to statement s_5 and s_6 (0.67 in Tarantula and 0.71 in Ochiai) instead of the real fault s_4.

We propose an iterative path clustering to aid fault localization, by introducing path vectors and calculating their angles in order to partition them into categories. In the first iteration, the 13 test cases in our example are divided into three failed groups ($\{t_1\}$, $\{t_2, t_4\}$, $\{t_3\}$) and four passed groups. This classification may not only improve statement ranking, but also helps to reduce the size of test cases. So we select a fraction of the classified test cases to pinpoint faulty statements, without any loss of accuracy. On the other hand, classification information is further used, not only for describing test case distribution, but also for having a better understanding of labeled groups. Assertions, called **path descriptors**, are used to describe the execution paths associated with test cases. So a failed group is distinguished from its neighbor (the nearest passed group), especially in next iterations. And a better understanding of program faults is gained by comparing the logical difference in path descriptors of a failed group and of the nearest passed one – a statement at the deviation point is first to be suspected. In our example, failed group $\{t_1\}$ is labeled by r=y while its neighbor $\{t_5, t_6\}$ by r<y, which turns a spotlight on the real culprit s_4.

Next section will present our framework of fault localization based on fuzzy *c*-lines clustering algorithm, and illustrate fault understanding by comparing path descriptors obtained by iterative and interactive queries.

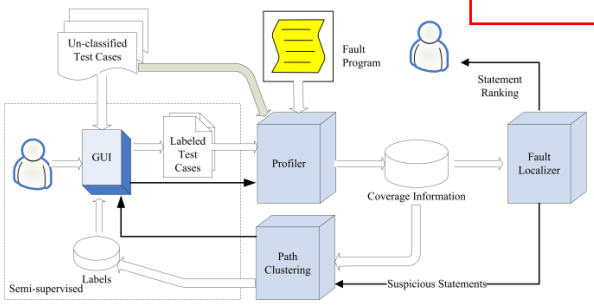

Figure 1. Framework of the proposed approach

III. FUZZY CLUSTERING FOR FAULT LOCALIZATION

A. Description of the Approach

The approach we propose is a framework as shown in Fig.1. At the conceptual level, it works on a fault program (pictured by a yellow sheet) and a bunch of test cases, in a loop of profiling, clustering and fault pinpointing. Initially, unclassified test cases and the faulty program are fed into the profiler to produce coverage information about executed statements,

which are afterwards used by a fault localizer to pinpoint faulty statements, and by path clustering to label test cases also. The outgoing gray arrows, starting from un-classified test cases, means that their place will be taken over by the labeled ones in the next round of profiling when path clustering was initially set up. Please note that ongoing path clustering can be interactively controlled by users via a graphical user interface, as will be described algorithmically in sub-section C.

修改

B. *Fuzzy Path Clustering*

In fuzzy clustering (also referred to as soft clustering), data elements can belong to more than one cluster, and associated with each element is a set of membership levels. Fuzzy c-means (FCM) [9] is one of the well-known fuzzy clustering algorithm and many FCM modifications have been proposed after FCM. A first group of modifications is based on shapes of clusters. Linear fuzzy clustering algorithms such as fuzzy c-lines (FCL) [10] and fuzzy c-varieties (FCV) [11] were constructed to treat linear dates and capture linear substructures replacing the prototypical Fuzzy c-means.

捕获

原型的

In FCL clustering algorithm [10], the lines in R^S through a point v with unit direction vector d are denoted as $L(v; d) = \{y \in R^S \mid y = v + td; t \in R\}$. For sparse source separation, because the sensors signals will approximately form some lines through the origin in m-dimensional geometric space, we can only consider a special FCL clustering: the lines in R^S are all through the origin ($v = 0$), and they can be denoted as $L(d) = \{y \in R^S \mid y = td; t \in R\}$. Before formally describing path clustering, we present some notations to be used next.

Definition 1. (Feature Point, *FP*) A feature point refers to a special statement in a program, denoted by *i.a*, where *i* is the line number of the statement, and *a* a variable, which denotes either the number of execution of line *i*, or a non-empty assertion derived from the expression on line *i*.

Definition 2. (Path) A path is a program run, i.e. a sequence of statements executed by a test case, denoted by $<i_1.a_1, i_2.a_2, \ldots, i_n.a_n>$.

Definition 3. (Path Vector) A path vector is a subset of a path $<i_1.a_1, i_2.a_2, \ldots, i_n.a_n>$, consisting of values of feature points, denoted by $<a_{j1}, a_{j2}, \ldots, a_{jl}>$.

路径描述符号

Definition 3. (Path Descriptor) A path descriptor is the name of a path (vector), denoted by a non-false conjunction $a_j \wedge a_k \wedge \ldots \wedge a_l$ of assertions in the path.

产生

Example 1. Consider program qr(x,y) in Table I, t_1 yields a path *<1.1, 2.1, 3.1, 4.1, 4.*r=y, *4.*r<y, *5.0, 6.0, 7.1>*, and it can have multiple path vectors, for example, a full path vector is *<1, 1, 1, 1, 1, 0, 0, 0, 1>* where *4.*r=y becomes true and *4.*r<y becomes false after running statement s_4 with t_1. A shorter path vector can be *<1, 1, 1, 0>*, referring to the *<3.1, 4.1, 4.*r=y, *4.*r<y> part. And the path associated with t_1 is described by '*4.*r=y'.

The user decides to choose a short or long path vector; he/she may choose full path at the beginning, and shorter path as the loop of profiling, clustering and fault pinpointing goes, in which the user may revise the path vector based on the current statement ranking returned by the fault localizer.

Provided with path vectors, we think that all the vectors in the same direction will make a line that goes through the origin. If we regard those vectors in the same line as one cluster, they come from the same linear equation. So suppose data pairs in $S = \{(x_{1h}, \ldots, x_{mh})^T \mid h = 1, \ldots, N\}$ are drawn from c different hyper-lines: $L_i = \{x \in R^m \mid x = tb_i ; t \in R\}$, $i = 1, \ldots, c$ or

$$\frac{x_1}{b_{1i}} = \cdots = \frac{x_m}{b_{mi}}, i = 1, \cdots, c \tag{1}$$

where $x_h = (x_1, \ldots, x_m)^T \in R^m$ is the variable vector and $b_i = (b_{1i}, \ldots, b_{mi})^T$ is the unit direction vector of L_i.

The distance from $x_h = (x_{1h}, \ldots, x_{mh})^T$ to L_i is

$$\begin{aligned} d(x_h, L_i)^2 &= \| x_h - z^* \|^2 \\ &= \| ox_h \|^2 - \| oz^* \|^2 = < x_h, x_h > - (< x_h, b_i >)^2 \end{aligned} \tag{2}$$

where o is the origin, z^* is on line L_i and $x_h z^* \perp L_i$.

Similar to the FCM, the objective function of our path clustering is

$$J_m(U, b) = \sum_{h=1}^{N} \sum_{i=1}^{c} (u_{ih})^m d(x_h, b_i)^2 \tag{3}$$

where $m \in [1, +\infty)$ is the weighting exponent.

Minimization of such an objective function in (3) yields simultaneous estimates for the directions of the c hyper-lines, together with a fuzzy c-partition of the data. First, the same as FCM, we get fuzzy c-partition matrix U as

$$u_{ih}^{(r+1)} = \begin{cases} \left[\sum_{j=1}^{c} \left(d(x_h, b_i) / d(x_h, b_j) \right)^{2/(m-1)} \right]^{-1}, & I_h = \varnothing \\ 1/n_h, & I_h \neq \varnothing, i \in I_h \\ 0, & I_h \neq \varnothing, i \notin I_h \end{cases} \tag{4}$$

where $I_h = \{i \mid 1 \le i \le c, d(x_h, b_i) = 0\}$ and n_h is the number of elements in I_h. Second, for fixed U and x, minimizing the (3) is equivalent to maximize

$$\xi_i(b_i) = \sum_{h=1}^{N} (\hat{u}_{ih})^m (< b_i, x_h >< x_h, b_i >) = b_i^T \left(\sum_{h=1}^{N} (\hat{u}_{ih})^m x_h x_h^T \right) b_i, \quad i = 1, \cdots, c \tag{5}$$

Note that b_i ($i = 1, \ldots, c$) is an eigenvector of

$$S_i = \sum_{h=1}^{N} (\hat{u}_{ih})^m x_h x_h^T, \quad i = 1, \cdots, c \tag{6}$$

corresponding to its largest eigenvalue.

The *path clustering algorithm PC* is executed as follows: At the beginning of algorithm *PC*(P, T), parameters are initialized by assigning the number of the clusters c ($1 \le c \le n$), setting the weighting exponent $m > 1$, specifying the cluster representation as (3), defining $d(x_h, b_i)^2$ as (2), picking a termination threshold $\varepsilon > 0$, and setting an initial partition $U^{(0)}$ to be empty and iteration index r to be 0. In calculating the c model parameters b_i, we will find the eigenvalues based on (6)

and the eigenvector b corresponding to the largest eigenvalue (i.e. $b_i = b$). Termination of looping is checked in some convenient induced matrix norms at line 4.

Algorithm 1: *PC*(*P*, *T*)

Inputs: a program *P*, test cases *T*.

Output: labeled test cases $U^{(i)}$

1 initialize parameters *c*, *m*, ε and *r*

2 calculate the *c* model parameters b_i

3 $U^{(r+1)} \leftarrow U^{(r)}$ with $d(x_h, b_i)^2$ as define in (2)

4 **if** $\| U^{(r)} - U^{(r+1)} \| \le \varepsilon$ **then** exit; **else** $r \leftarrow r+1$;

5 goto line 2.

Note that the number of clusters is usually decided with the aid of a *cluster validity criterion* system. Many cluster validity criteria have been proposed for fuzzy clustering, but most of them were designed in point-wise cluster instead of hyper-line-shaped representation. Next we introduce a new one for our path clustering algorithm.

We know that the inner product of two real unit vectors $<b_i, b_j>$ equals the projection of b_i on the space spanned by b_j, $<b_i, b_j>$ is the projection length, and the only factor that influences the projection length is the angle between them. We use this value to measure the difference between two hyper-lines that pass through the origin ($v = 0$) point in that a zero projection length implies that the two hyper-planes are orthogonal, while a unit projection length implies their coincidence. Since all c hyper-lines pass through the origin and each of them is featured by a corresponding b_i with unit length, we can easily judge the difference of two hyper-lines by an absolute value of a standard inner-product of their unit normal vector $|<b_i, b_j>|$. From the discussion above we see that, the more diverse the clusters are, the larger the separation validity function will be. This fits the concept of separation measure criterion. Motivated by Xie and Beni's separation function [12], we define a new one suitable for the hyper-line-shaped representation:

$$F = f_{com} / f_{sep} = \sum_{h=1}^{N} \sum_{i=1}^{c} u_{ih}^{m} d(x_h, b_i)^2 \Big/ N\left(\min_{i<j} \left(\left| \langle b_i, b_j \rangle \right| + k \right)^{-1} \right) \quad (7)$$

where *k* is a rather small real positive constant that prevents the function from being divided by zero, and the numerator is a compactness function, which reflects the compactness of clusters, and the denominator is the separation function, which indicates the separation of clusters.

In our framework, we plot (F vs. *c*) in the GUI to display a significant change in curve and thus help the user choose an appropriate number of clusters.

C. Iterative Diagnosis with Path Descriptors

Our framework features diagnosing faults iteratively and interactively. As shown in Fig. 1, ongoing path clustering is controlled by users via a GUI, we describe this procedure in Algorithm 2, which will work in either an automatic or semi-supervised mode. Lines 1~5 capture the typical run of *PCFL* (*P*, *T*, ∅) in an automatic mode, where *PCFL* processes all test cases *T* for once, and returns an initial clustering. Semi-supervised mode goes on with the initial clustering but works in a controllable way; the user invokes $GUI(U^{(i+1)}, SS^{(i)}, F^{(i)})$ to graphically display test case labels, path descriptors and suspicious statements. Profiling, clustering and fault pinpointing repeatedly, the user can improve his/her understanding of real faults by comparing the path descriptor of a failed execution and that of its nearest passed execution, and of course stop this cycle.

Algorithm 2: *PCFL* (*P*, *T*, *FP*)

Inputs: a program *P*, test cases *T*, the set *FP* of current feature points.

Output: labeled test cases $U^{(i)}$, path descriptors $PD^{(i)}$ and suspicious statements $SS^{(i)}$

1 $U^{(0)} \leftarrow T$, $F^{(0)} \leftarrow FP$, $i \leftarrow 0$

2 $PD^{(0)} \leftarrow descriptor(P, F^{(0)})$

3 $CI^{(0)} \leftarrow profiling(P, U^{(0)}, F^{(0)})$

4 $SS^{(0)} \leftarrow localizing(CI^{(0)})$

5 $U^{(i+1)} \leftarrow PC(CI^{(i)}, F^{(i)})$

6 $F^{(i+1)} \leftarrow GUI(U^{(i+1)}, SS^{(i)}, F^{(i)})$

7 **if** $F^{(i+1)}$ is empty **then** exit; **else** $i \leftarrow i+1$;

8 $CI^{(i)} \leftarrow profiling(P, U^{(i)}, F^{(i)})$

9 $SS^{(i)} \leftarrow localizing(CI^{(i)})$

10 goto line 5

IV. EXPERIMENTS

We developed the proposed framework, by implementing the profiler and the fault localizer in C and the path clustering component in Matlab. These components read their inputs and write outputs in text files.

To evaluate our approach, we chose the single fault case in the Siemens test suite, which is the most famous benchmark for validating the effectiveness of fault localization techniques. The test suite contains seven programs, their faulty versions and thousands of test cases. The first hypothesis is that our approach gains from reduced program runs without loss of accuracy, while the second is that it can improve TBFL in isolating and understanding program errors. To verify these two hypotheses, we carried two studies as follows:

A. Study One

To support the first hypothesis, we classified a whole bunch of the Siemens test suite, and obtained the number of paths before and after clustering, as shown in Figure 2.

In Figure 2, the test cases of Print_tokens, Print_tokens2, Replace, Schedule, Schedule2, Tcas and Tot_info are categorized into 15, 13, 12, 10, 11, 12 and 9 groups, which account for 0.97% on average, of the whole test cases. So a great gain in efficiency is achieved from reduced coverage information.

Also we run TBFL techniques such as Tarantula and Ochiai with the reduced test cases, and found that the accuracy of fault localization can be ensured when the number of passed test cases are 3-5 times more than failed ones.

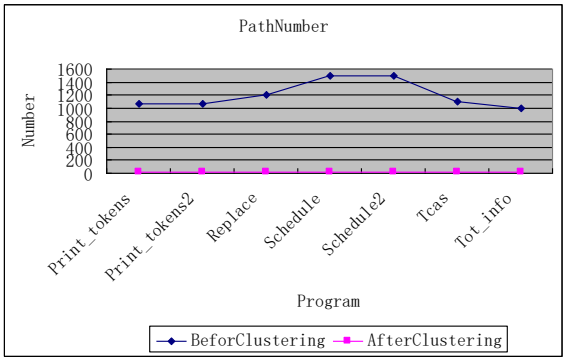

Figure 2. Comparison of the Efficiency

B. *Study Two*

In this study, we selected 20 participants from undergraduates enrolled in data structure courses at the Dalian Maritime University. These students have knowledge of C language. They are not skilled programmers but have no problem of reading and debugging C programs like the Siemens test suite. All participants were given an 8-hour training on the TBFL tool in our framework, by thinking a description of the failure in Print_tokens and the way of reproducing it, and identifying the fault causing such failure. After that they were individually asked to localize and understand the fault causing failures in other fault programs, for which they are separated into two groups based on their performance: both are ensured to have the same performance on average, group (A) work only with the TBFL tool while a controlled group (B) work with the whole framework in a semi-supervised mode.

TABLE II. FAULT LOCALIZATION(L) AND UNDERSTANDING(U)

Program	**Ver.**	**L**		**U**	
		(A)	(B)	(A)	(B)
Print_tokens2	2	0:27	0:21	0:44	0:16
Replace	12	1:20	1:00	1:41	0:53
Schedule	9	1:16	0:57	1:15	1:00
Schedule2	10	0:55	0:23	1:11	0:45
Tcas	11	1:28	1:00	1:15	0:51
Tot_info	15	1:36	1:14	1:19	0:46

For the tasks assigned to group (A) and (B), the average completion time h:m, in hours (h) and minutes (m), is listed in Table II, where we did not consider the time for the initial path clustering. From Print_tokens (Schedule) to its variant Print_tokens2 (Schedule2), we find that time in localizing and understanding the fault is reduced sharply due to better program understanding. In all cases, path clustering aided Tarantula and Ochiai to have gains in terms of time, because the reduced coverage information revs up fault localization. The possible threat to the validity is the right choice of FPs, so Group (B) were taught to determine next-round FPs by watching more suspicious statements in the GUI, and to stop when there is no obvious difference between two consecutive iterations. We observed that the completion times of the two groups in understanding the fault are quite different, and the students in group (B) were more efficient in explaining the faulty statement causing the failure because the difference in path descriptors makes sense in understanding bug sources.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we present an improved fault localization method based on a fuzzy c-Lines clustering algorithm. We use the Siemens suite to validate the effectiveness of our method, and most of the faulty version in Siemens suite contains a single fault. Experimental results show that the proposed method has achieved good efficiency in fault localization and understanding on the Siemens test suite.

For the future work we plan to extend our framework with run-time program dependences. Also we will do more experiments with multi-bugs programs in the near future.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No.61175056, NO.61402070), the Natural Science Foundation of Liaoning Province of China (No. 2015020023), the Educational Commission of Liaoning Province of China (No. L2015060) and the Fundamental Research Funds for the Central Universities (NO.3132016348).

REFERENCES

[1] W. E. Wong, and V. Debroy. Software Fault Localization. IEEE Reliability Society 2009 Annual Technology Report.

[2] J. A. Jones, M. J. Harrold, J. Stasko. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002). pp. 467–477. ACM, NY, USA.

[3] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In Automated Software Engineering (ASE), pages 30–39, 2003.

[4] J. A. Jones, M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2005). pp. 273–282.

[5] M. Weiser. Programmers use slices when debugging. Communica-tions of the ACM pp. 446–452, 1982.

[6] A. Zeller, R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 28(2), 2002.

[7] Y. Miao, Z. Chen, S. Li, Z. Zhao, Y. Zhou. Identifying Coincidental Correctness for Fault Localization by Clustering Test Cases. SEKE 2012: 267-272

[8] Z. Gao, Z. Chen, Y. Feng, B. Luo. Mining Sequential Patterns of Predicates for Fault Localization and Understanding, SERE 2013.

[9] F. Hoppner, F. Klawonn, R. Kruse and T. Runkler. Fuzzy Cluster Analysis. Jhon Wiley and Sons, 1999.

[10] J. C. Bezdek, C.Coray, R. Gunderson and J. Watson. “Detection and characterization of cluster substructure I. Linear structure fuzzy c-lines ” SIAM Jour. of Appl. Math, 40:2 (1981), 339-357.

[11] J. C. Bezdek, C. Coray, R. Gunderson and J. Watson. “Detection and characterization of cluster substructure II. Fuzzy c-Varieties and convex combinations thereof” SIAM Jour. of Appl. Math, 40:2 (1981), 358-372.

[12] X.L. Xie and G.A. Beni, “Validity measure for fuzzy clustering,” IEEE Trans. Pattern and. Machine Intell., vol. 3, no. 8, pp. 841-846, 1991.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 15–26, New York, NY, USA, 2005. ACM.

[14] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. SIGSOFT Softw. Eng. Notes, 30:286–295, September 2005.

[15] A. Groce. Error explanation with distance metrics. In TACAS, volume 2988 of Lecture Notes in Computer Science. Springer, 2004.

[16] B. Demsky, M. Rinard. Automatic detection and repair of errors in data structures. ACM SIGPLAN Notices, 38(11):78--95, 2003.

[17] F. Wotawa. Debugging vhdl designs using model-based reasoning. AI in Engineering, 14(4):331--351, 2000.