Towards an Emerging Theory for the Diagnosis of Faulty Functions in Function-call Traces

Syed S. Murtaza and Abdelwahab Hamou-Lhadj
Software Behaviour Analysis Research Lab
ECE, Concordia University
Montreal, QC, Canada
{smurtaza, abdelw@ece.concordia.ca}

Nazim H. Madhavji
Dept. of Computer Science,
University of Western Ontario,
London, Ontario, Canada
madhavji@csd.uwo.ca

Mechelle Gittens
Dept. of Computer Science
University of West Indies
Cave Hill, Barbados
mechelle.gittens@cavhill.uwi.edu

***Abstract*—When a fault occurs in the field, developers usually collect failure reports that contain function-call traces to uncover the root causes. Fault diagnosis in failure traces is an arduous task due to the volume and size of typical traces. Previously, we have conducted several research studies to diagnose faulty functions in function-call level traces of field failures. During our studies, we have found that different faults in closely related functions occur with similar function-call traces. We also infer from existing studies (including our previous work) that a classification or clustering algorithm can be trained on the function-call traces of a fault in a function and then be used to diagnose different faults in the traces where the same function appears. In this paper, we propose an emerging descriptive theory based on the propositions grounded in these empirical findings. There is scarcity of theorizing empirical findings in software engineering research and our work is a step towards filling this gap. The emerging theory is stated as: a fault in a function can be diagnosed from a function-call trace if the traces of the same or a different fault in that function are already known to a clustering or classification algorithm. We evaluate this theory using the criteria described in the literature. We believe that this emerging theory can help reduce the time spent in diagnosing the origin of faults in field traces.**

***Index Terms*—Fault diagnosis, Function-call traces, Field failures, Corrective software mainteance**

I. INTRODUCTION

Software maintainers use failure reporting techniques to collect information about system failures in the field. Usually the failure reports consist of configuration information and function-call traces from the field. Windows Error Reporting tool[1], the Mozilla crash reporting system[2], IBM DB2 [23], and IBM WebSphere [15] systems are examples of reporting system that collect function-call traces for crashing and non-crashing failures. However, failure reports of software products with a large client base can be quite overwhelming to software developers due to their sheer volume. This problem is further aggravated by the large size of typical traces [16]. It is known to be a challenging task to analyze the content of these traces. It is no surprise that fault diagnosis in corrective software maintenance can take up a large amount of the maintenance time.

In previous work, we have conducted three different studies to help developers in quickly diagnosing faulty functions in function-call level failure traces of deployed software applications [25][26][27]. We have found that function-call traces of different faults in closely related functions occur with similar function-call traces: if a classifier (e.g., decision tree) is trained on the traces of one fault in a function, it can then be used to diagnose different faults in failure traces of that function. We extensively evaluated this approach on 12 different programs ranging from 1000 LOC to 20 million LOC in different empirical settings. These programs include seven programs in Siemens suite [16], a Space program [11], three UNIX utilities [11], and a large industrial program from IBM [27]. Our experiments showed that faults in functions of approximately (average) 85% of the function-call level failure traces of an application can be diagnosed by using decision trees.

Prior researchers have focused on clustering function-call traces of failures from the deployed software systems to group together similar faults [5][9][10][22][30]. We also evaluated clustering technique in a similar manner to our classification approach and found that faults in functions of approximately 77% of the failure traces can be diagnosed.

Thus, we observed that our empirical research findings can lead to the foundation of an emerging theory in the field of software fault diagnosis for function-call level traces. We, therefore, propose an emerging descriptive theory that describes the relationship between function-call traces of different faults in a bottom-up fashion. First, we define the propositions of the emerging theory based on our empirical findings. The propositions of the emerging theory are formed by a *hypothetico-inductive* model as described by Sjoberg et al. [31]. Second, we use the propositions to state the emerging theory. Third, we evaluate the propositions using the criteria for measuring the goodness of a theory as described by Sjøberg et al. [31]. In short, we state the emerging theory based on the propositions derived from our empirical findings as follows:

> *A fault in a function can be diagnosed from a function-call trace if the traces of the same or a different fault in that function are already known to a clustering or classification algorithm.*

We believe that this emerging theory will facilitate the diagnosis of the origin of faults in function-call traces, particularly during software maintenance. It will also reduce

1http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx
[2]http://crash-stats.mozilla.com

the time and effort spent in corrective software maintenance and will lead to improvements in software quality: developers can spend more time in fixing the faults rather than finding them. To our knowledge, the emerging theory is novel and no such work has been undertaken on an emerging theory in Software Engineering for fault diagnosis in function-call traces. Stol et al. noted that there is a need to effectively theorize research findings in the field of software engineering [32], and our work contributes in that direction.

The paper continues as follows. The details of our three studies are described in Section II. The propositions of emerging theory are formed in Section III and the emerging theory is stated in Section III using those propositions. The propositions are evaluated in Section IV and the implications of this theory are discussed in Section V. The related work is described in Section VI and the paper is concluded in Section VII.

II. BACKGROUND

Figure 1 shows an overview of our approach in our earlier work. The key objective of our earlier work was to diagnose faulty functions in a program's function-call based failure traces collected from the field after deployment. To diagnose faulty functions in new failure traces, we used historical failure traces of that program. The historical failure traces were composed of past failure traces collected from the field for the program, or the failure traces of artificial faults (mutants) seeded automatically in that program. The faulty functions were also known in the historical traces. We then trained multiple decision trees on the historical traces using one-against-all approach [33]. In the one-against-all approach, a dataset of multiple labels (in our case multiple faulty functions) is divided into multiple dataset such that all the failure traces of one label (i.e., faulty function) are labelled as faulty and the traces of all other faulty functions are labelled as *others*. This allowed us to trained one decision tree on the failure traces of every faulty function—i.e., one decision tree per faulty function. Whenever a new failure trace arrived, we passed it to the trained decision trees which in turn predicted their labels, the faulty function or the *others* label, with a probability. We arranged all the predicted faulty functions (excluding others) in the decreasing order of their probabilities with the intuition that the function ranked higher in the list would be more likely to faulty. The ranked list was then presented to the developers. This process is also shown in Figure 1.

Figure 1 also shows examples of function-call traces where a function entry represents when control enters the function and function exit represents when control exits it. For example, consider that a fault occurred in a function *foo4* because of an invalid input—foo4 requires a number and string was passed. A trace for this fault has been collected from the field and passed on to our technique. Our technique compares this trace against the historical set of existing traces of different faulty functions by using the decision trees. If there are traces for the same fault (i.e., invalid input) of *foo4* or any other fault (e.g., divide by zero overflow) in *foo4*, then the decision tree trained for *foo4* is going to predict the faulty function *foo4*. All the predicted faulty functions by different decision trees are then arranged in the decreasing order of the probabilities and presented to a developer. Final evaluation is done by measuring how many functions a developer needs to review to find the correct faulty function.

Figure 1 actually shows the general overview of our approach for our previous three studies [25][26][27]. Three studies were performed in different empirical settings using this process and each of them with their specific process is briefly explained in the following sections. We have also described evaluation of clustering techniques in the following section.

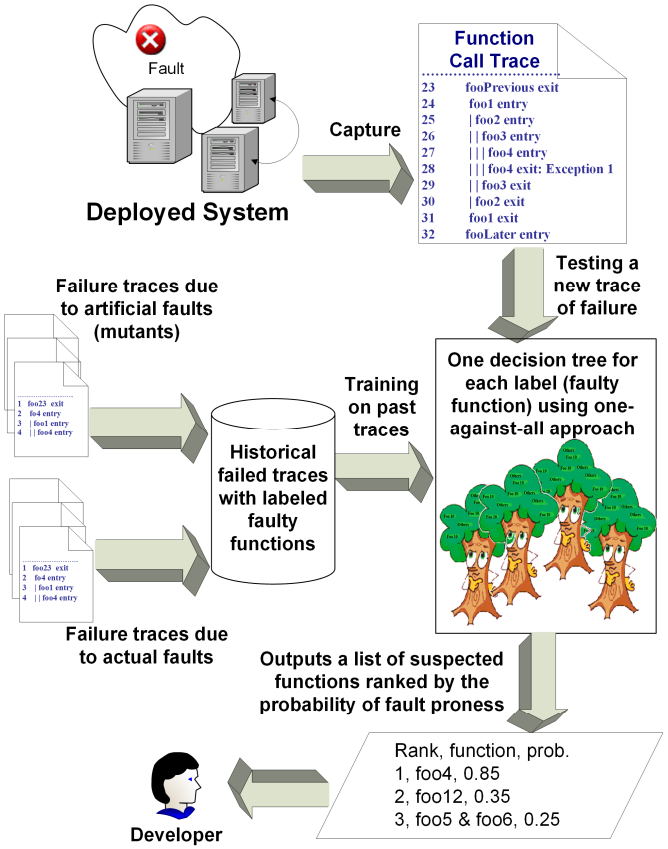

Fig. 1. Overview of our approach

A. *Study [S1]: Finding Recurrent Faults from the Field using Function-call level Failure Traces of Software in the Field [26]*

The objective of the first study was to diagnose the recurrent faulty function in the function-call level traces of field failures [26]. In practice, 50-90% of the failures are the rediscoveries of previous faults [22][27][34]. We call them recurrent faults. The focus of this study was on those field failures that occurred due to recurrent faults or due to new faults in the recurring faulty function. The first study actually showed that function-call traces of the same or different faults in a faulty function are the same, and the failure traces of one fault can be used to diagnose another fault in the same function.

In the first study, we performed experiments on the Siemens suite, which is a collection of seven programs of approximately 1000 LOC each [11][16], and the Space program, which constituted approximately 10,000 LOC. There were 7-21 functions in the seven programs of Siemens suite and 136 functions in the Space program. The number of failure traces was 275-4266 and the number of distinct faulty

functions was 2-10 in the Siemens suite. The number of failure traces in the Space program was 71,958 and the number of distinct faulty functions was 26. In the Siemens suite, the faults were hand seeded by several developers independently [11], and in the Space program faults were found during development at the European Space Agency [11]. The failure traces were collected by executing test cases. The failure traces of these programs consisted of both crashing and non-crashing failures. A non-crashing failure is more difficult to diagnose than the crashing failure because a fault in the non-crashing failure can occur long before the appearance of the failure [30]. Our approach focused on both kinds of failures.

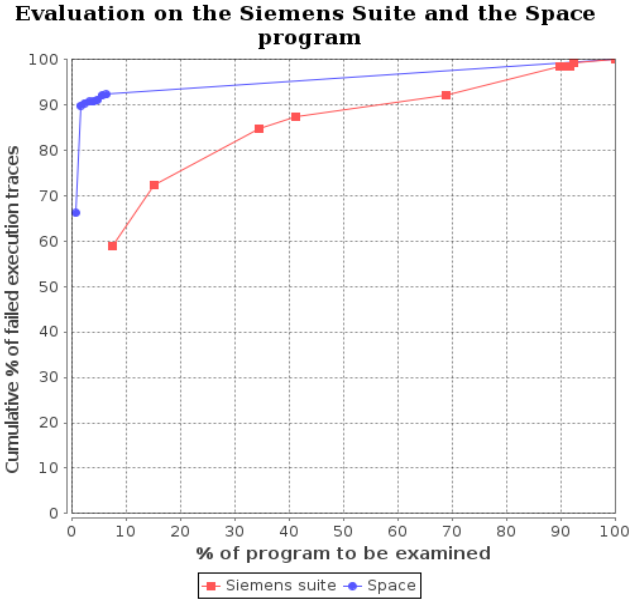

Fig. 2. Diagnosis of faulty functions in the failure traces of the Siemens Suite and Space Program

We partitioned the dataset of failure traces into 3 stratified parts; only one part (33%) of the data was kept for training and the rest (66%) was kept for testing. This is to simulate the realistic case of minimal set of traces available in the field. To evaluate that the different faults in the same functions occur with similar function-call traces, we only kept failure traces of one fault in a faulty function in the training set and put the failure traces of different faults in a function in the test set. The decision trees are then trained on the traces in the training set and tested on the traces in the test set.

In Figure 2, we show the results on the seven programs of the Siemens suite and the Space program. The horizontal axis (X-axis) represents the percentage of a program that needs to be examined by a developer before diagnosing the faulty function from the ranked list of functions generated by our technique for a test trace. It is measured by the number of functions reviewed by a developer up to the faulty functions from the ranked list of functions divided by the total number of functions. The vertical axis (Y-axis) measures the cumulative percentage of system failure traces that achieve a score within a segment on X-axis. This metric is an effective way of assessing the results, also adopted by other fault localization techniques [8][20], and it is given in **Equation 1**.

$$\begin{bmatrix} \%\ of\ program \\ to\ review \end{bmatrix} = \frac{Functions\ reviewed\ upto\ the\ faulty\ function}{Total\ functions} * 100$$

Equation 1. Estimating program review effort in functions

For example, a point (2, 90) on the chart in Figure 2, can be read as only 2% of the program (number of functions) was required to be reviewed by developers to diagnose faulty functions in 90% of the failure traces. In the case of Space program, 2% of the program was equivalent to 1.4 functions and in the case of the Siemens suite 14% of the program was equivalent to 2 functions (i.e., 1 function = 7% of the program). This showed that recurrent faulty functions irrespective of the type of faults can be diagnosed in the function-call level failure traces easily by using a classifier and by reviewing only first few functions.

B. Study [S2]: Identifying Recurring Faulty Functions in Field Traces of a Large Industrial Software System[27]

In the second study, we extended the evaluation of our approach on a large industrial program of 20 million lines of code (LOC) and 200,000 functions[3] [27]. The objective of this study was to find recurrent faults in the failure traces of a large industrial program collected from the field and address the issue of scalability on traces exceeding several Gigabytes. During the evaluation, we have determined that different types of events in function-call traces do not contribute to the automated diagnosis of faulty functions using a classifier. For example, exceptions thrown and functions that occur consistently without much variation across traces can be ignored.

We were able to collect crashing and non-crashing failure traces of three different releases, for a period of three years, for this large commerical application. The types of the faults varied from crashing faults (e.g., null pointer) to non-crashing faults (e.g., performance faults). All the failure traces that we collected were actually caprtured when a failure manifested itself to customers of this application in the field.

In a similar manner to study [S1] (Section II.A), we divided the trace dataset of each release into 33% training set and 66% test set. The results for the evaluation of our approach on each of the three releases are shown in Figure 3. In this large program, the total number of functions were approximately 200,000 and the average number of distinct functions in a trace were 10,000. However, we considered that the total number of functions reviewed by the programmer to diagnose the faulty function would be approximately 1000 at least (after discussing with programmers of this large system). Therefore, in Figure 3, a point (0.8, 79) of the code review means 8 functions are required to review for diagnosing the faulty functions in 79% of the failure traces.

In this large software system, there were 82% recurrent faults in our sample dataset, so the accuracy we obtained (i.e. 65% to 80% depending on the release) is out of the 82% existing recurrent faults. This is equivalent to an accuracy of 92% (Release 1), 98% (Release 2), and 80% (Release 3) out of 100% recurring faulty functions. The average accuracy is

[3]Due to confidentiality reason, the name of the commercial software product is not disclosed.

therefore 90% on the review of 0.8% or less of the code (fewer than 8 functions). A line (without markers) in Figure 3 shows that no classification is made, and a developer has to use 100% of the program to identify faulty functions. This line goes up to the last point (100,100) on the chart but it is not shown up to 100% on the chart for better visibility of the markers. Mostly, the line represents those traces that have newer faulty functions and are not found in the training set.

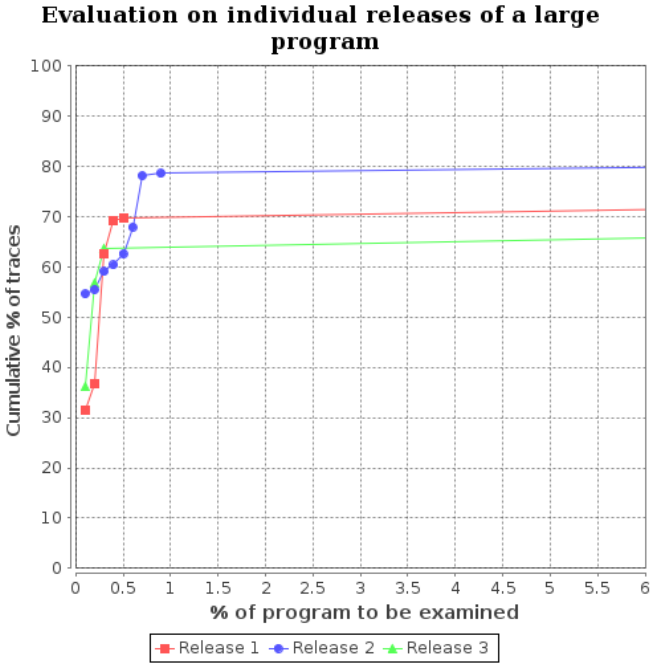

Fig. 3. Diagnosis of faulty function in the failure traces of three releases of an industrial application of 20 million LOC

In this study, we also evaluated our approach by training the decision trees on the failure traces of previous releases and using them to diagnose faulty functions in the failure traces of the succeeding releases. In Figure 4, we show the results of identification of recurring faulty functions in traces of succeeding releases by training decision trees on earlier releases. Figure 4 shows that faulty functions in approximately 97% of the failure traces were diagnosed by reviewing 3% to 4% of the program when training was done on previous releases. In short, this shows that recurrent faulty functions can be diagnosed across releases irrespective of the type of fault.

In this study, we also focused on the work done by other researcher. Prior researchers focusing on traces of deployed software systems have mostly focused on clustering function-call traces of failures from the field [5][9][10][22][30]. The majority of these techniques focus on clustering traces of crashing failures [5][9][10][22]. They usually form clusters by measuring similarity in function-call sequences of top frames of stacks (functions that execute last). Podgurski et al. [30] proposed a technique of k-medoid clustering for non-crashing failures. The authors' intuition was that traces in each cluster would belong to the same faulty file. Non-crashing failures are difficult to diagnose than the crashing failures because a fault may not appear in functions appearing in the collected function-call trace.

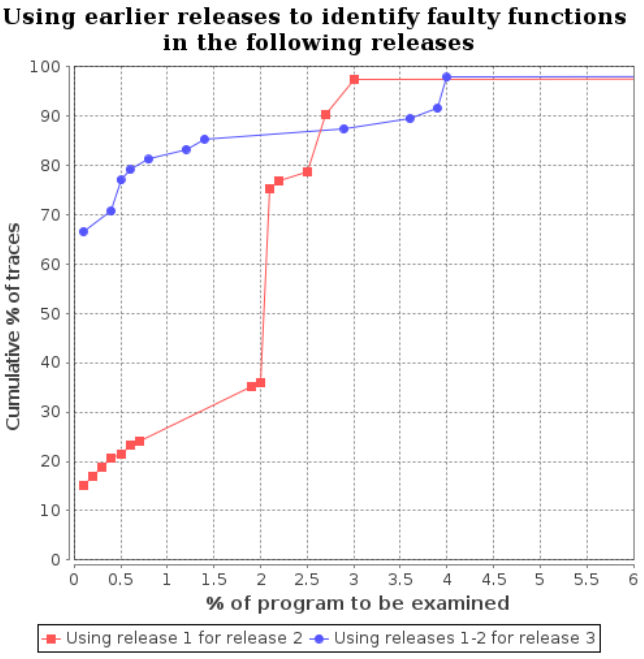

Fig. 4. Diagnosis of faulty functions across releases in an industrial application of 20 million LOC

In order to evaluate clustering based approaches, we implemented a clustering technique ourselves. We used the dataset of traces of a large software system of 20 million lines of code, already described in Section II.B. In our dataset, the traces were a combination of both crashing and non-crashing failures. Therefore, we used k-medoid clustering on the function-call traces of the large software system, a similar approach to Podgurski et al. [30]. We applied clustering on the traces of one the release of the large software system by forming as many groups (clusters) as there were classes (faulty functions) in training traces. We employed Manhattan distance as the median based distance measure by using Weka [33]. The idea was that each cluster would represent one class (i.e., faulty functions). We found out that many clusters contained traces of more than one faulty function. To evaluate clustering exactly in the same way as our approach, a ranking method was required such that closely related clusters for a new trace in the test set can be predicted in an ordered list. No such ranking method exists in the literature.

We created a clustering based ranking on the basis of simple intuition. First, we clustered traces in the training set using k-medoid clustering with as many clusters (groups) as there were faulty functions. Second, we measured the Manhattan distance of a trace in the test set to all the formed clusters and assigned the trace to a cluster with a minimum Manhattan distance. Third, we matched the faulty function of the test trace with one of the *m* faulty functions of the cluster that the trace was assigned to. If a match was found, then we considered that *m* functions were reviewed by a developer to discover the faulty function. Fourth, in the case of no match, we matched the faulty function of the trace with the faulty functions of other clusters one by one in decreasing order of the number of traces in the clusters. The intuition is that the developer would consider one of the faulty functions of the cluster with the largest number of traces as the suspected faulty function for the trace. When the match is not found,

the developer would review faulty functions of the cluster with the second largest number of traces, and so on, to the last cluster. Fifth, the effort of the developer was measured by Equation 1, the number of functions reviewed up to the diagnosis of actual faulty function of the trace. In a similar manner to our approach, we considered the total number of functions as 1000 for Equation 1. Figure 5 shows the results of clustering based ranking on Release 1 of the large industrial program. Figure 5 also shows the results of our approach on the same release.

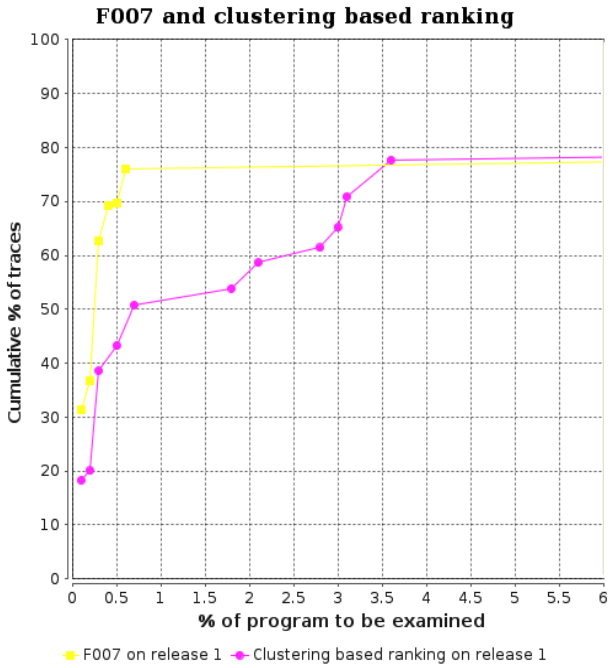

Fig. 5. Our approach (called F007) against an approach using ranking based on k-medoid clustering

Figure 5 shows that our approach diagnoses faulty functions in function-call traces with a better accuracy than a clustering-based approach. However, Figure 5 also shows that the clustering-based approach does not yield poor results. In fact, faulty functions in 77% of the function-call traces can be diagnosed by reviewing 3.5% of the program using the clustering-based approach. The traces in the training set had the same or different faults in functions when compared to the traces in the test set, and the clustering based approach was still able to diagnose faulty functions in up to 77% of the traces in the test set. This shows that there is a certain degree of similarity in function-call traces of different faults in a function.

C. *Study [S3]: An Empirical Study on the Use of Mutant Traces for Diagnosis of Faults in Deployed Systems[25]*

In the third study, we investigated how artificial faults, generated using software mutation in test environment, can be used to diagnose actual faults in deployed software systems. The use of traces of artificial faults can provide relief when it is not feasible to collect different kinds of traces from deployed systems. In this study, we also investigated the similarity of function-call traces of different faults in functions using artificial and actual faults [25].

In this study, we first generated mutants (artificial faults) for every function of a program. A software mutant is an artificially generated fault in a program and Andrews et al. [1] showed that mutants are close representative of actual faults. In the next step, we executed test cases on these mutants and collected traces when the test cases failed. We called these traces *mutant traces* and labeled them with the corresponding faulty function. We then trained decision trees on these mutant traces and used them to diagnose faulty functions in the actual failure traces. The actual failure traces were also collected by running test cases. We evaluated this approach on the three UNIX utilities, namely Grep, Gzip and Sed [11], and on the Space program developed at the European Space Agency [11]. The results are shown in Figure 6 and it can be interpreted in the same way as earlier results for the studies [S1] and [S2].

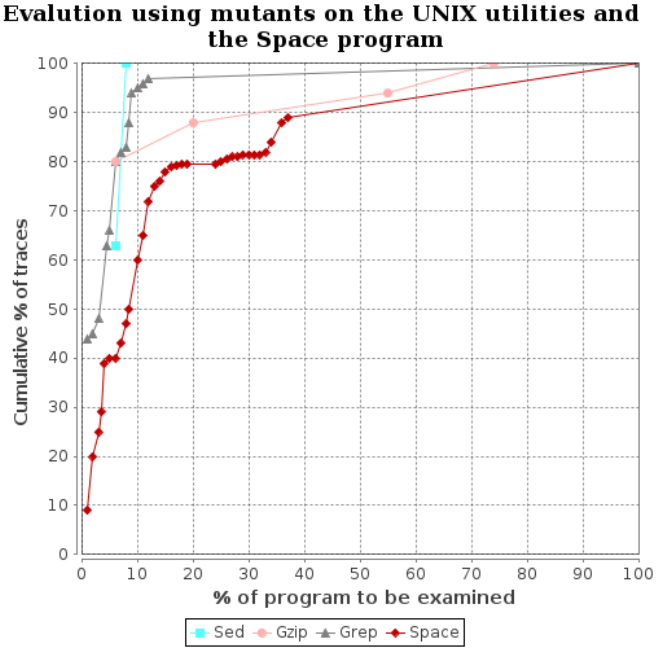

Fig. 6. Diagnosis of faulty functions in actual failure traces of four programs by using traces of mutants (artificial faults)

The results in Figure 6 show that faulty function in 80-100% of the failure function-call traces can be diagnosed by reviewing 5-20% of the program's functions. These results show that different faults in the same functions have similar function-call traces because mutant faults and actual faults are different faults. However, the results also show that function-call traces of some faulty function overlap with some other faulty functions because 100% accuracy was not obtained on the review of first function.

III. Emerging Theory

Sjøberg et al. [31] proposed a framework for describing Software Engineering (SE) theories by using the frameworks proposed earlier for other sciences, such as social sciences [24][35] and information systems sciences [6]. Sjøberg et al. argue that Software Engineering (SE) theories are different from social and behavioral sciences because SE theories are more applied and dependent on time and place at the current stage of development [31]. For example, change in education and skill of a software engineer over time may change the validity of a theory. Also the context of lab or industry as a placeholder may affect the validity of a theory [31]. Thus, we adopted the framework for describing SE theories by Sjøberg et al. to propose an emerging theory in this paper.

Sjøberg et al. identified three levels of abstractions to develop a theoretical proposition. In the first level (or Level 1), relationships that are concrete and can be directly inferred from the observations become the Level 1 propositions. Level 2 propositions are abstract representation of possibly many Level 1 theoretical propositions. Finally, Level 3 theoretical proposition combine all other theoretical propositions and tend to articulate an aspect of Software Engineering (SE).

In Table I, we hierarchically organize different level of propositions emerging from our earlier studies described in Section II. Table I uses the labels [S1], [S2] and [S3] to refer to the studies described in Section II. Level 1 proposition is observed directly from the empirical results of the studies. Level 2 proposition is the higher level abstract representations of Level 1 propositions. Both Level 1 and Level 2 propositions are testable and tested in their source studies. Each level proposition is also assigned a unique number which will be used in explaining the propositions below. There are no Level 3 propositions for our findings because typically Level 3 findings are derived from a larger set of studies as the discipline becomes mature [31].

A. Proposition P^1_1

During our experiments on the Siemens suite [16] and the Space program [11] in the study [S1], we have observed that F007 can identify the same faulty functions with different faults in 60-90% of the failure traces on reviewing 2 to 3 functions in the Space program and the Siemens suite, respectively (see Section II.A). For the Siemens suite, 7% of the program-review was approximately equivalent to 1 function, and for the Space program 1% of the program-review was equivalent to 1 function. This implies proposition P^1_1 and it shows that different faults in the same function occur with similar occurrences of function calls but up to a certain limit.

B. Proposition P^1_2

Our experiments on a large commercial program of 20 million LOC showed that recurring faulty functions in up to 90% of the failure traces from the field can be easily diagnosed on the review of less than 1% of the program. This is when a classifier is trained on the traces of same or different faults in the same functions. This facilitated in quickly diagnosing recurrent faults and implying the proposition P^1_2 from the study [S2].

C. Proposition P^1_3

In study [S2], we found that common faulty functions exist in multiple releases of a program, and common faulty functions in failure traces of a succeeding release can be diagnosed by using the failure traces of those functions from prior releases. When a classifier was trained on the failure traces of earlier releases, it was able to diagnose common faulty functions in up to 98% failure traces of future releases on the review of approximately 4% of the program. This implies proposition P^1_3 and it again shows a similarity in function-call traces of different faults in the same functions.

D. Proposition P^1_4

In order to determine how different or similar are the function calls of faults in one function with the function calls of faults in other functions we conducted the study [S3]. In the study [S3], we made every function in the Space, Grep, Gzip and Sed programs artificially faulty using mutants. The results showed that faults in the actual failure traces can be diagnosed using traces of artificial faults (i.e., proposition P^1_4).

E. Proposition P^1_5

Several researchers proposed the use of clustering on function-call traces of deployed software systems in the past. When we applied clustering on function-call traces of the large program of 20 million LOC in study [S2], we again found out similarity in function-call traces of faults in a function. We were able to diagnose faulty functions in 78% of the failure traces of a large program by reviewing approximately 3.5% of the program. This implies proposition P^1_5.

F. Proposition P^2_1

The Level 1 propositions imply that the faulty functions in the actual failure traces are not entirely distinguishable but they are distinguishable up to a certain extent. This is because 100% or closer accuracy was not obtained on reviewing first function from the ranked list of faulty functions. However, near 100% accuracy was still obtained on the review of first few functions in the ranked list (i.e., a small percentage of a program). This can be observed from all the Level 1 propositions P^1_1 to P^1_5.

In all the propositions, except P^1_4, only those functions were used as faulty functions that were actually found (or made) faulty in development or field. One can argue that there were fewer faulty functions (as labels or groups) for classifier or clustering algorithm, therefore the higher accuracy was obtained on the review of few functions (labels in terms of machine learning). However, in the proposition P^1_4, we made all the functions faulty in a program and collected their failed (mutant) traces. In the case of P^1_4, we still observed that the accuracy is still not poor; i.e., review of all the functions (100% of the program) is still not required to diagnose faulty functions in majority of the traces. In terms of machine learning, there are many more labels for P^1_4 than all other propositions but still they accuracy is similar. This implies that function-call traces of faults in a function overlap with the function-call traces of faults in some other functions but at the same time the function-call traces of faults in a function are also different from traces of many other functions. The reason is that if function-call traces of all the functions had overlapped, then we would have had to review about 100% (or closer to 100%) of the program (functions) to identify the faulty functions in any trace. However, we found faulty functions in traces by reviewing only few functions.

TABLE I. THEORETICAL PROPOSITIONS ARISING FROM THE EMPIRICAL STUDIES.

[S1], [S2] and [S3] represent the three studies described in Section II.

P^L_J represents a unique proposition number, where L = level number and J=proposition id at that level.

Level 1 proposition	Level 2 proposition
(P^1_1) Faulty functions in 60-90% of failure traces can be diagnosed on reviewing 2-7% program, when a classifier is trained on the traces of at most one fault in those functions of the Space program and the Siemens suite [S1].	*(P^2_1)* A group 'Mi' of related functions has similar function-call traces when a fault occurs in any of the functions of that group 'Mi', and function-call traces of 'Mi' are different from the function-call traces of another group of function 'M_k' if a fault occurs in the functions of group 'M_k'; where i, k= 1-n and $i \neq k$ and Mi $\subset$ N and $M_k \subset$ N and N={functions | functions $\in$ program}. If only traces of one fault in a function are already known, then that faulty function in unknown failure traces can be diagnosed due to the similarity of function calls of a group.
(P^1_2) Faulty functions in up to 90% of failure traces can be diagnosed on reviewing 0.8% program, when a classifier is trained on the traces of same or different faults in those functions of the large commercial program of 20 million LOC and 200,000 functions [S2].	
(P^1_3) Faulty functions in 98% of failure traces in a succeeding software release can be identified on reviewing 4% of the program, when a classifier is trained on the traces of faults in the same functions of preceding software releases of a large program of 20 million LOC and 200,000 functions [S2].	
(P^1_4) Faulty functions in 80-100% of the actual failure traces can be diagnosed on reviewing 5-20% of the program, when a classifier is trained on the traces of mutants (artificial faults) of all the functions in the Grep, Gzip, Sed and Space programs [S3].	
(P^1_5) Faulty functions in 78% of the actual failure traces can be diagnosed on reviewing approximately 3.5% of the program, when a clustering approach was applied to the function-call traces of a large program of 20 million LOC and 200,000 functions [S2].	

We can generalize this to proposition P^2_1 that there are M groups of closely related functions, and functions in each group make calls to each other or call the same functions regularly. When a fault occurs in one of the functions of a group (e.g., M_i), then function calls overlap with each other. When a fault occurs in a function in another group M_k, then there are fewer overlapping function calls with the function calls of faults in groups other than M_k. Thus, the faulty function in an unknown failure trace can be diagnosed by only knowing the failure traces of one (same or different) fault in that function (e.g., as in P^1_1) but this diagnosis will require reviewing few functions. Few functions will be required to review because a function-call trace of a fault in function overlaps with traces of some other functions that form a group. This results in proposition P^2_1

G. Emerging Theory Statement

The proposition P^2_1 generalizes from the propositions at Level 1 by using the fact that a faulty function can be approximately identified from traces if a classification or clustering algorithm is trained on the traces of at least one (same or different) fault of that faulty function. Thus based on the propositions at Level 1 and Level 2 we state the emerging theory as:

A fault in a function can be diagnosed from a function-call trace if the function-call traces of the same or a different fault in that function are already known to a clustering or classification algorithm.

IV. EVALUATING THE EMERGING THEORY

Sjøberg et al. [31] also list criteria for evaluating the "goodness" of theories. Similar criteria to measure the goodness of theories were also presented by Boehm and Jain (2005). We have adopted the following criteria from the work of Sjøberg et al. In fact, Sjøberg et al. criteria are almost similar to the work of Boehm and Jain [4] criteria and can be considered as their representative. Sjøberg et al. (and also Boehm and Jain [4]) criteria were adapted for SE theory evaluation from other disciplines such as Business Management [2], Psychology [14], and Sociology [7].

Following are the criteria taken from the work by Sjøberg et al. (2008) where each criterion designates the degree of support (i.e., low, medium, or high) for the emerging theory from the empirical studies (e.g., [S1][S2] and [S3] in our case) The classification of each criterion as low, medium, or high is based on a researcher's subjective judgment. Our judgment is derived from the explanation given by Sjøberg et al. for each criterion and it is further explained in the following subsections.

A. Empirical Support

The first criterion is the degree to which a theory is supported by empirical studies that confirm its validity [31]. We consider that empirical support will be high if the evaluation of a theory is done using a series of studies that complement each other; whereas, empirical support will be low if there is only one study that evaluates the technique. The reason is that if there are many studies repeating the same evaluation of a theory, then we can consider that the results of this theory will be the same in practice.

The empirical support of this emerging theory is considered to be medium because the number of programs on which we performed experiments in three studies was 12 from small to very large programs and four of these programs had several releases (see Section II). This shows that the results were empirically grounded in the results from a sufficient number of programs and from different types of experiments in three studies. Thus, we consider that the empirical support is medium and there is certainly room to do more.

B. Utility

The second criterion determines the degree to which a theory supports the relevant areas of the software industry [31]. We consider that the utility of a theory will be high if the propositions of a theory can be used as input in decision making, understanding and prediction in a given industrial setting. The utility of a theory will be low if the theory is not able to reduce the complexity of the empirical world and decision making.

The emerging theory can be used as an aid to maintainers in identifying the origin of faults during software maintenance. During maintenance approximately 50-90% of the faults in the field are rediscoveries of previously known faults [22][27][34]. Maintainers can use our technique, discussed in the lead text in Section II, to diagnose faulty functions in the field failures with a lesser time and effort than the contemporary techniques. Thus, we consider utility of this theory to be high in practice.

C. Generality

The third criterion determines the breadth of the scope of a theory and the degree to which the theory is independent of specific settings [31]. We consider that higher generality means broader applicability of a theory in different settings; whereas, lower generality means application of a theory is valid in specific settings.

We have experimented on 12 different programs of small to very large sizes in different experimental settings: we consider this theory to be generalizable to other programs. In 11 of the programs, the traces were collected in lab settings by running test cases. In the case of the very large program, traces were actually field traces and were collected when failure occurred at the customer site. The theory itself is independent of specific formats and program elements in a trace, making it more generalizable to systems across different programming concepts (such as process-to-process communication mechanisms, events, triggers, message passing, call/return, etc.). This theory is also independent of a programming language and the age of a program because we analyzed execution traces not the constructs of source code to discover faulty functions. We also evaluated several releases of programs, a large program with legacy and new code, and the programs written by several hundreds or thousands of developers. Considering all these aspects we judge medium generality for this theory.

D. Testability

The fourth criterion determines the degree to which a theory can be empirically refuted [31]. We consider higher testability when propositions of a theory are internally consistent, free from ambiguities, and tested in empirical studies. Alternatively, we consider lower testability when all propositions of a theory are not tested in empirical studies and the propositions lack consistency such that they are not easy to be tested in other replicated studies.

The propositions of the emerging theory are defined in a consistent, understandable and non-ambiguous way. Each of the studies [S1][S2][S3] can be easily replicated and the stated propositions (P^1_1, P^1_2, P^1_3, P^1_4, P^1_5 and P^2_1) are based on these studies. Each of the propositions has been empirically validated and tested. Different study designs (e.g., identifying faults at system's configuration level) can be used to independently test the propositions. We consider the testability high for this emerging theory.

E. Explanatory Power

The fifth criterion identifies the degree to which a theory accounts for and predicts all known observations within its scope. The fifth criterion determines that the theory is simple in that it has few ad hoc assumptions and relates to that which is already well understood [31]. We judge that a theory will have high explanatory power when it can be supported by analogies to well-known theories, explains all relevant relationships, and accounts for all known data in its field. Alternatively, we consider explanatory power low for a theory when it cannot be associated with well-known theories and misses some relationships in its explanation.

The emerging theory presented in this paper provides an explanation of identification of a faulty function in a failure trace from traces of different faults in that function. We believe that theory can be made stronger in explanatory power by identifying quantitative characteristics to its attributes. For example: (a) what proportion of failure traces can be resolved correctly? (b) can we generalize this theory quantitatively like the 80-20 Pareto rule [3][13][29] (e.g., can we resolve 80% of failure traces using traces of 20% of functions)? Thus, we consider explanatory power low for this emerging theory, and further studies can strengthen the explanatory power of this theory.

F. Parsimony

The sixth criterion determines the degree to which a theory is economically constructed with minimum of concepts and propositions. There is a delicate balance between parsimony and explanatory power. We consider that higher parsimony means removal of unnecessary concepts and propositions that add little additional value to our understanding; whereas, lower parsimony means complex concepts and propositions that are difficult to understand.

This emerging theory at both levels of proposition is constructed using few, clear and concise concepts (such as function calls, traces, faults, program-review and faulty

functions). The applications of these concepts were also shown in Section II. Thus we think that parsimony is high.

V. Implications of the Emergning Theory

The emerging theory has several implications on both research and practice.

A. Research

Researchers can validate this theory by using it as a preliminary hypothesis and by performing experiments on different programs or from different perspectives. The results can then be fed back to this theory and it can be modified or further strengthened.

Researchers can further build new emerging theories based on this theory, for example, by investigating what is the relationship among the functions in a group, how different functions form a group when a fault occurs in them, and what are those functions? Such theories could be used to determine the groups of functions before releasing software and traces of a fault in one function can be used to identify another faulty function of the same group.

Moreover, researchers can investigate a new theory using the 80-20 Pareto rule for software code [3][13][29] as a basis. For example, if 20% of the code is causing 80% of the faults, then is it possible to identify faulty functions in 80% of the traces using the traces of 20% functions?

B. Practice

The emerging theory will have its implications in improving the quality of software. Software quality will improve because the maintainers can spend more time on fixing the faults rather than diagnosing the faults. The emerging theory will also help in reducing the time and effort spent in corrective maintenance. It can be used in diagnosing faults in configuration of a system using operating system level call traces. It can also be used in diagnosing fault location during the testing phase of succeeding releases using the failure traces of previous releases. In addition, the theory can be further applied to sample traces or abstract views of function-call traces (see [17][18] for examples of trace abstraction techniques), relieving software engineers from having to examine large-sized traces.

VI. Related Work

Software Engineering (SE) research is usually not guided by explicit theories and nor does it yield explicit theories. Researchers within the SE community have also argued that SE needs strong theoretical foundation to become a real engineering science [32]. Only few authors have built the guidelines to construct SE theories by borrowing insights from other disciplines, such as Sjøberg et al. [31]. Sjøberg et al. proposed a framework to develop constructs and proposition of SE theories and to evaluate the goodness of SE theories (see Section III and Section IV). Using the Sjøberg et al. criteria, Ferrari [12] also developed an emerging SE theory on the interaction of system architecting and requirement engineering. Ferrari performed a sequence of empirical studies and used the empirical findings to define an emerging theory. Our work in this paper is inspired by the work of Ferrai [12]. Similarly, Lawrance et al. [21] proposed an information foraging theory for how programmers perform debugging during software development. Lawrance et al. [21] also used the criterion defined by Sjøberg et al. as the basic building block of their theory. Stol and Fitzgerald [32] argued that SE research papers does show the traces of an SE theory and they called them theory fragments. They use a framework from social science on three papers having high impact in SE to illustrate the role of theorizing in SE research. They conclude that SE researchers already theorize but they just don't know yet and new researchers need training to conduct a theory focused research.

In this paper, we have taken a step towards the creation of an emerging theory for the diagnosis of faulty functions from function-call traces. We proceeded by theorizing the empirical findings of the three studies we conducted earlier [25][26][27] and the studies conducted by other researchers [5][9][10][22][30] on function-call traces from the field. To our knowledge, this work is novel and as such it contributes to the SE theories.

VII. Conclusion

Diagnosis of faults from traces of field failures is an difficult and time consuming task. In this paper, we propose an emerging theory on the diagnosis of faults from function-call level traces of field failures by using empirical findings of existing studies (see Section II). This emerging theory identifies the relationship between function-call traces of different faults in a function. It is stated as:

A fault in a function can be diagnosed from a function-call trace if the traces of the same or a different fault in that function are already known to a clustering or classification algorithm.

The emerging theory was developed in a bottom-up fashion using a *hypothetico-inductive* model [31] and each of its propositions (see Section III) was empirically grounded in the findings of the three studies (see Section II). Subsequently, we also evaluated this theory (see Section IV) on the basis of "theory goodness" criteria proposed by Sjøberg et al. [31] and similar criteria of Boehm and Jain [4]. Overall, the emerging theory satisfies the goodness criteria of utility, generality, parsimony, testability, empirical support and explanatory power (see Section IV).

This theory is still at its initial stages as it was derived from experimenting with only 12 programs of small (1000 LOC) to very large sizes (20 million LOC). Clearly, more empirical studies are needed to test the specific aspects of the theory, such as what are those functions that form a group of functions with the similar function-call traces. More efforts are needed from the maintenance community to conduct studies in various contexts and from various perspectives to strengthen or validate this emerging theory.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an Appropriate Tool for Testing Experiments?," In *Proc. of the 27th International Conference on Software Engineering,* ACM, St. Louis, USA, 2005, pp. 402-411.

[2] S. B. Bacharach, "Organizational Theories: Some Criteria for Evaluation," *The Academy of Management Review*, vol. 14, no. 4, 1989, pp. 496–515.

[3] B. Boehm, and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer,* vol. 34, no. 1, IEEE CS Press, 2001, pp. 135-137.

[4] B. Boehm, and A. Jain, "An Iinitial Theory of Value-based Software Engineering," In *Value-Based Software Engineering (1st edition)*, Springer, LNCS, Germany, 2005, pp. 15-33.

[5] M. Brodie, Ma Sheng, G. Lohman, L. Mignet, M. Wilding, J. Champlin, P. Sohn, "Quickly Finding Known Software Problems via Automated Symptom Matching," In *Proc. of the 2nd International Conference on Autonomic Computing*, Seattle, WA, 2005, pp. 101-110.

[6] J. Carroll, and P. A. Swatman, "Structured-case: A Methodological Framework for Building Theory in Information Systems Research," *European Journal of Information Systems*, vol. 9, no. 4, 2000, pp. 235–242.

[7] B. Cohen. *Developing Sociological Knowledge: Theory and Method.* 2nd ed., Belmont, CA: Wadsworth Publishing, 1989.

[8] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," In *Proc. 19th European Conference. on Object-Oriented Programming*, Springer LNCS, Glasgow, UK, 2005, pp. 528-550.

[9] Y. Dang, R. Wu, H. Zhang, D. Zhang, P. Nobel, "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity," In *Proc. of the 34th International Conference on Software Engineering ,* Zurich, Switzerland, 2012, pp. 1084-1093.

[10] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox," In *Proc. of 27th IEEE International Conference on Software Maintenance*, 2011, pp. 333-342.

[11] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, Springer, 2005, pp. 405-435.

[12] R. Ferrari, "An Emerging Theory on the Interaction Between Requirements Engineering and Systems Architecting based on a Suite of Exploratory Empirical Studies," *Ph.D. dissertation, University of Western Ontario,* Ontario, Canada, 2010.

[13] M. Gittens, Y. Kim, and D. Godwin, "The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software." In *Proc. of the 29th International Conference on Computer Software and Applications*, Edinburgh, Scotland, 2005, pp. 179-185.

[14] B. D. Haig, "An Abductive Theory of Scientific Method," *Psychological Methods*, vol.10, no. 4, 2005, pp. 371–388.

[15] Hare, D.; and Julin, D.; "The Support Authority: Interpreting a WebSphere Application Server Trace File", *IBM WebSphere Developer Technical Journal*, 2007. http://www.ibm.com/developerworks/websphere/techjournal/0704_supauth/0704_supauth.html

[16] A. Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension", *Ph.D. Dissertation, School of Information Technology and Engineering (SITE),* University of Ottawa, 2005.

[17] H. Pirzadeh, A. Hamou-Lhadj, "A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension," In *Proc. of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '11),* pp. 221 - 230, 2011.

[18] A. Hamou-Lhadj, and T. Lethbridge, "Reasoning About the Concept of Utilities," *ECOOP International Workshop on Practical Problems of Programming in the Large, Oslo, Norway, Lecture Notes in Computer Science (LNCS),* vol. 3344, Springer-Verlag, pp. 10-22, 2004.

[19] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, "Experiments on The Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria," In *Proc. of the 16th International Conference on Software Engineering*, IEEE CS, Sorrento, Italy, 1994, pp.191-200

[20] J. A. Jones, and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," In *Proc. of 20th International Conference on Automated Software Engineering*, IEEE/ACM, CA, USA, 2005, pp.273-282.

[21] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S. D. Fleming, "How Programmers Debug, Revisited: An Information Foraging Theory Perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, 2013, pp.197,215.

[22] I. Lee, and R. Iyer, "Diagnosing Rediscovered Problems Using Symptoms, " *IEEE Transactions on Sofware Eng*ineering, vol. 26, no. 2, 2000, pp.113-127.

[23] R. B. Melnyk, "DB2 Basics: An introduction to the DB2 UDB trace facility," *DB2 Information Development, IBM Canada Ltd.*, 2004. http://www.ibm.com/developerworks/data/library/techarticle/dm-0409melnyk/index.html

[24] R. K. Merton. *Social Theory and Social Structure*, 3rd ed., New York: The Free Press, 1968.

[25] S. S. Murtaza, A. Hamou-Lhadj, N. H. Madhavji, M. Gittens, "An Empirical Study on the Use of Mutant Traces for Diagnosis of Faults in Deployed Systems", *Journal of Systems and Software*, Elsevier, vol. 90, 2014, pp.29-44.

[26] S. S. Murtaza, M. Gittens, Z. Li, N. H. Madhavji, "F007: Finding Rediscovered Faults from the Field Using Function-level Failed Traces of Software in the Field", In *Proc. of the 2010 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, ACM, Canada, 2010, pp. 61-75.

[27] S. S. Murtaza, N. H. Madhavji, M. Gittens, A. Hamou-Lhadj, "Identifying Recurring Faulty Functions in Field Traces of a Large Industrial Software System," *IEEE Transactions on Reliability,* vol. 64, no. 1, 2014, pp. 269-283.

[28] A. Offutt, and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," In *Mutation Testing for the New Century*, USA: Kluwer Academic Publishers, 2001, pp. 34-44.

[29] T. J. Ostrand, E. Weyuker and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Eng*ineering, vol. 31, no. 4, 2005, pp. 340-355.

[30] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, B. Wang, "Automated Support for Classifying Software Failure Reports," In *Proc. of 25th International Conference on Software Engineering*, IEEE CS, Portland, US, 2003, pp. 465-475.

[31] D. Sjøberg, D. Dyba, B. C. Anda, and J. Hannay, "Building Theories in Software Engineering," In *Guide to Advanced Empirical Software Engineering*, London: Springer, 2008, pp. 312–336.

[32] K. J. Stol, B. Fitzgerald, "Uncovering Theories in Software Engineering," In *Proc. of 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE)*, 2013, pp.5-14.

[33] I. H. Witten, and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann*, San Francisco, USA, 2005.

[34] A. Wood, "Software Reliability from the Customer View," *Computer*, vol. 36, no. 8, IEEE CS, 2003, pp.37-42.

[35] R. K. Yin. *Case Study Research: Design and Methods*. Thousand Oaks, CA, USA: Sage Publications, 1984.