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

Syed Shariyar Murtaza, 1,2Abdelwahab Hamou-Lhadj, 1,3Nazim H. Madhavji, 1,4Mechelle Gittens

1,2Dept. of Computer Science, University of West Indies, Cave Hill, Barbados
3,4Dept. of Computer Science, University of Western Ontario, London, Canada
1,2Dept. of Electrical and Computer Engineering, Concordia University, Canada

smurtaza@encs.concordia.ca, 2,3abdelw@ece.concordia.ca, 3,4madhavji @csd.uwo.ca, 4,3mechelle.gittens@cavhill.uwi.edu

Abstract—When a fault occurs in the field developers usually collect failure reports (traces) containing function-call traces. (A function-call trace is a sequence of functions executed by a program.) Fault diagnosis in failure traces collected from the field is an arduous task due to the volume and size of trace contents. Previously, we have conducted several research studies to diagnose faulty functions in function-call level traces of field failure. During our studies, we have found that different faults in closely related functions occur with similar function-call traces. We also discovered that our previous studies and studies from other researchers that a classification or clustering algorithm can be trained on the function-call traces of a fault in a function and then can be used to diagnose different faults in the traces of the same function. 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 research and our work is a step towards this direction. 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 in Section III. The propositions are evaluated in Section IV and the implications of this theory are discussed in Section V.

Keywords—fault diagnosis, function-call traces, failure reports, corrective software maintenance

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. Mozilla crash reporting system [1], IBM DB2 [14], and IBM WebSphere [10] systems are examples of reporting system that collect function-call traces for crashing and non-crashing failure. However, failure reports of software products with a large client base can be quite overwhelming to software developers due to their sheer volume and large sizes. It is no surprise that fault diagnosis in corrective software maintenance can take up 40-80\% of the maintenance time [19].

In previous work, we have conducted three different studies to facilitate developers in quickly diagnosing faulty functions in function-call level failure traces of deployed software applications [15],[16],[17]. We have found that function-call traces of different faults in closely related functions occur with similar function-call traces; and if a classification algorithm (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 function-call traces of that function. We extensively evaluated this approach on 11 different open source programs and a proprietary IBM system from 1000 LOC to 20 million LOC in different empirical settings. Our experiments showed that faults in functions of approximately (on average) 80\% of the function-call level failure traces of an application can be diagnosed by using the decision trees.

Most of the prior researchers have focused on clustering function-call traces of failures from the deployed software systems to group together similar faults [3],[5],[6],[13],[18]. 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 these 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 by using the criteria described by Sjoberg et al. [20]. We state the emerging theory as below.

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 fault in function-call traces. It will also reduce the time and effort spent in corrective software maintenance. Stol et al. noted that there is a need to effectively theorize research findings in the field of software engineering [21], and our work contributes in that direction.

The details of our and prior studies are described in Section II. The propositions of emerging theory and the emerging theory are stated in Section III. The propositions are evaluated in Section IV and the implications of this theory are discussed in Section V.
II. BACKGROUND

The key objective of our earlier works was to diagnose faulty functions in a program’s function-call based failure traces collected after its deployment. A faulty function is a function that is the cause of a fault in a function-call level failure trace. To diagnose faulty functions in new failed 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 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. 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 failed traces of one label (i.e., a 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 failed 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 label, 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.

Figure 1 shows the general overview of our approach for the three studies. 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. For details on comparison with the related techniques, readers are referred to individual studies.

A. Study [S1]: Finding Recurrent Faults from the Field using Function-level Failed Traces of Software in the Field [16]

The objective of the first study was to diagnose the recurrent faulty functions in the function-call traces of field failures [16]. In practice, 50-90% of the failures are the rediscoveries of previous faults [13], [17]. 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.

In the first study, we performed experiments on the Siemens suite, which is a collection of seven programs of approximately 1000 LOC each [7],[11], and the Space program, which constituted approximately 10,000 LOC. In the Siemens suite, the faults were hand seeded by several developers independently [7], and in the Space program faults were found during development at the European Space Agency [7]. The failure traces of these programs consisted of both crashing and non-crashing faults.
of failure traces that achieve a score within a segment on X-axis.

\[
\frac{\text{Functions reviewed up to the faulty function}}{\text{Total functions}} \times 100
\]

**Equation 1. Estimating program review effort in functions**

In Figure 2, in the case of the 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[17]**

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 from IBM [17].

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.

**Evaluation on individual releases of a large program**

![Figure 3. Evaluation on a program of 20 million LOC](image)

We were able to collect function-call level failure traces of three different releases, for a period of three years, for this large commercial application. The types of the faults varied from crashing faults (e.g., null pointer) to non-crashing faults (e.g., performance faults). 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 functions. However, we considered that the total number of functions reviewed by the programmer to diagnose the faulty function would be approximately 1000 at least. 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.

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 a previous release. In short, this shows that recurrent faulty functions can be diagnosed across releases irrespective of the type of fault.

**Using earlier releases to identify faulty functions in the following releases**

![Figure 4. Evaluation across releases of a large program](image)

In this study, we also focused on the work done by other researchers. Prior researchers focusing on traces of deployed software systems have mostly focused on clustering function-call traces of failures from the field [3],[5],[6],[13],[18]. The majority of these techniques focus on clustering traces of crashing failures [3],[5],[6],[13]. They usually form clusters by measuring similarity in function-call sequences of top frames of stacks (functions that execute last). Podgurski et al. [18] propose a technique of k-medoid clustering for non-crashing failures. Non-crashing failures are difficult to diagnose than the crashing failures because a fault may not appear in functions appearing last in a function-call trace.

To evaluate clustering exactly in the same way as our approach, 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. Fifth, the effort of the developer was measured by Equation 1. We considered the total number of functions as 1000 for Equation 1. Figure 5 shows the results of clustering based ranking and our approach on Release 1 of the large industrial program.

Figure 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 clustering based approach. However, the clustering based approach is still able to diagnose faulty functions in up to 77% of the traces in the test set on the review of 3.5% of the program. The traces in the training set had the same or different faults in functions when compared to 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[15]

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 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. [2] 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 failed traces. We evaluated this approach on the three UNIX utilities, namely Grep, Gzip and Sed [7], and on the Space program developed at the European Space Agency [7]. 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].

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

The results in Figure 6 shows that faulty function in 80-100% of the failed function-call traces can be diagnosed by reviewing 5-20% of the program. 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. [20] proposed a framework for describing software engineering (SE) theories by using the frameworks proposed earlier for social sciences and information systems sciences. 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 1, shown in Appendix A, we hierarchically organize different level of propositions emerging from our earlier studies described in Section II. Table 1 uses the labels [S1], [S2] and [S3] for 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 described 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 gets matured [20].

A. Proposition $P_1^1$

During our experiments on the Siemens suite [11] and the Space program [7] in the study [S1], we have observed that F007 can identify the same faulty functions with different faults in 60-90% of the failed traces on reviewing 1-3 functions in the Space program and the Siemens suite, respectively (see Section II.A). 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 [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. 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 and 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 failed traces can be diagnosed using traces of artificial faults (i.e., proposition $P_1^4$).

E. Proposition $P_2^1$

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 failed traces of a large program by reviewing approximately 3.5% of the program. This implies proposition $P_2^1$.

F. Proposition $P_2^2$

The Level 1 propositions imply that function-call traces of faults in a function overlap with the function-call traces of faults in some other functions. In fact, the results show 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., $M_1$) 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 $M_k$. 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 to identify the faulty functions in any trace. However, we could still find faulty functions in failed traces by reviewing only few functions. In fact, the Level 1 propositions show that fault in a function from an unknown failed trace can be diagnosed by only knowing the failed traces of the same or different fault in that function. This diagnosis may require reviewing few functions due to the similarity of function-calls in a group of functions. This results in proposition $P_2^2$.

G. Emerging Theory Statement

The proposition $P_2^2$ generalizes from the propositions at Level 1 by using the fact that a fault in a function can be approximately identified from traces when a classification or a clustering algorithm is trained on the traces of the same or different faults 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 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. [20] also list criteria for evaluating the “goodness” of theories. Each criterion designates the degree of support (i.e., low, medium, or high) for the emerging theory from the empirical studies. 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 explained below.

A. Empirical Support

First criterion is the degree to which a theory is supported by empirical studies that confirm its validity [20]. 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 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 (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.

B. Utility

The second criterion determines the degree to which a theory supports the relevant areas of the software industry [20]. 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 in 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 quickly identifying the origin of faults during software maintenance. This is because approximately 50-90% of the faults in the field are rediscoveries of previously known faults [13],[17]. 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 [20]. 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. The theory is independent of specific formats, program elements in a trace, a programming language, and the age of a program because we analyzed execution traces not the constructs of source code. 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. Thus, we judge medium generality for this theory.

D. Testability:

The fourth criterion determines the degree to which a theory can be empirically refuted [20]. 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 the propositions are not tested in empirical studies and 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. 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.

E. Explanatory Power:

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 [20]. 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 meet these criteria.

The emerging theory presented in this paper can be made stronger in explanatory power by identifying quantitative characteristics to its attributes. For example: (a) what proportion of failed traces can be resolved correctly? (b) can we generalize this theory quantitatively like the 80-20 Pareto rule [4],[9] (e.g., can we resolve 80% failed traces using traces of 20% functions)? Thus, we consider explanatory power low for this emerging theory as more studies are needed.

F. Parsimony

The sixth criterion determines the degree to which a theory is economically constructed with minimum of concepts and propositions. 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. Thus, 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 EMERGING THEORY

The emerging theory has several implications on both research and practice. We explain these implications below.

A. Research

Researchers can validate this theory by using it as a preliminary hypothesis, and by performing experiments from different perspectives. The results can then be used for modification or strengthening of this theory.

Researchers can further build new emerging theories based on this theory; e.g., 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? These 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 [4],[9]. 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 facilitate 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 failed traces of previous releases.
VI. RELATED WORK

Researchers within the SE community have also argued that SE needs strong theoretical foundation to become a real engineering science [21]. Only few authors have built the guidelines to construct SE theories by borrowing insights from other disciplines, such as Sjøberg et al. [20]. Using the Sjøberg et al. criteria, Ferrari [8] 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. Similarly, Lawrance et al. [12] proposed an information foraging theory for how programmers perform debugging.

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 create theories.

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 tried to theorize the empirical findings of the three studies we conducted earlier [15],[16],[17] and the studies conducted by other researchers[3],[5],[6],[13],[18] on function-call traces from the field. Our work is novel and contributes to the SE theories which otherwise are lacking.

VII. CONCLUSIONS

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 our studies and prior researchers. This emerging theory identifies that faults in a function can be identified if traces of the same or different faults in that function are already known to a classification or clustering algorithm. This theory is still in initial stages as it was done on only thirteen programs of small (1000 LOC) to very large sizes (20 million LOC). Clearly, more empirical studies are needed from the maintenance community to test in detail the specific aspects of the theory, such as is it possible to identify faults in functions of 80% of the traces by using faults in 20% of the functions.

REFERENCES

Table 1: Theoretical propositions arising from the empirical studies

<table>
<thead>
<tr>
<th>Level 1 proposition</th>
<th>Level 2 proposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P₁₁) Faulty functions in 60-90% of failure traces can be diagnosed on reviewing 1-7% 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].</td>
<td>(P₂₁) 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 ‘Mk’ if a fault occurs in the functions of group ‘Mk’; where i,k= 1-n and i ≠ k and Mi ⊂ N and Mk ⊂ N and N={functions ( \in ) program}. If only traces of one fault in a function are already known, then that faulty function in unknown failed traces can be diagnosed due to the similarity of function-calls of a group.</td>
</tr>
<tr>
<td>(P₁₂) 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].</td>
<td></td>
</tr>
<tr>
<td>(P₁₃) 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].</td>
<td></td>
</tr>
<tr>
<td>(P₁₄) 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].</td>
<td></td>
</tr>
<tr>
<td>(P₁₅) 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].</td>
<td></td>
</tr>
</tbody>
</table>