3-25-2011 12:00 AM

Finding Faulty Functions From the Traces of Field Failures

Syed Shariyar Murtaza, *The University of Western Ontario*

Supervisor: Dr. Nazim H. Madhavji, *The University of Western Ontario*
Joint Supervisor: Dr. Mechelle S. Gittens, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science

FINDING FAULTY FUNCTIONS FROM THE TRACES OF FIELD FAILURES

(Spine title: Finding Faulty Functions from the Function-call Level Traces)

(Thesis format: Integrated Article)

by

Syed Shariyar Murtaza

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Supervisor

_____
Dr. Nazim H. Madhavji

Co-supervisor

_____
Dr. Mechelle S. Gittens

Examiners

_____
Dr. Michael W. Godfrey

_____
Dr. Luiz Capretz

_____
Dr. Charles X. Ling

_____
Dr. James H. Andrews

The thesis by

Syed Shariyar Murtaza

entitled:

Finding Faulty Functions from the Traces of Field Failures

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

_____
Date

_____
Chair of the Thesis Examination Board

Abstract

Corrective maintenance, which rectifies field faults, consumes 30-60% time of software maintenance. Literature indicates that 50% to 90% of the field failures are rediscoveries of previous faults, and that 20% of the code is responsible for 80% of the faults. Despite this, identification of the location of the field failures in system code remains challenging and consumes substantial (30-40%) time of corrective maintenance. Prior fault discovery techniques for field traces require many pass-fail traces, discover only crashing failures, or identify faulty coarse grain code such as files as the source of faults. This thesis (which is in the integrated article format) first describes a novel technique (F007) that focuses on identifying finer grain faulty code (faulty functions) from only the failing traces of deployed software. F007 works by training the decision trees on the function-call level failed traces of previous faults of a program. When a new failed trace arrives, F007 then predicts a ranked list of faulty functions based on the probability of fault proneness obtained via the decision trees. Second, this thesis describes a novel strategy, F007-plus, that trains F007 on the failed traces of mutants (artificial faults) and previous faults. F007-plus facilitates F007 in discovering new faulty functions that could not be discovered because they were not faulty in the traces of previously known actual faults. F007 (including F007-plus) was evaluated on the Siemens suite, Space program, four UNIX utilities, and a large commercial application of size approximately 20 millions LOC. F007 (including the use of F007-plus) was able to identify faulty functions in approximately 90% of the failed traces by reviewing approximately less than 10% of the code (i.e., by reviewing only the first few functions in the ranked list). These results, in fact, lead to an emerging theory that a faulty function can be identified by using prior traces of at least one fault in that function. Thus, F007 and F007-plus can correctly identify faulty functions in the failed traces of the majority (80%-90%) of the field failures by using the knowledge of faults in a small percentage (20%) of functions.

Keywords

Faulty Function, Execution Trace, Software Maintenance, Decision Tree, Mutation, Fault Rediscovery, 80-20 Pareto Rule, Deployed Software, Cost Sensitive Machine Learning.

Co-Authorship Statement

This thesis is written in an integrate-article format. There were several co-authors of the papers (published or submitted) related to this thesis. The author of this dissertation is the primary author of all the publications arising out of this dissertation. Dr. Nazim H. Madhavji is a co-author of all the publications in the capacity of a supervisor of the research conducted. Similarly, Dr. Mechelle Gittens is also a co-author of all the publications in the capacity of a co-supervisor. From the inception of the idea of the F007 technique, their role has been active in carving out the technique for the impact on the body of knowledge. Zude Li is also a co-author of some of the papers in the capacity of his help in building a mathematical model for the pattern mining technique used in the paper, and providing feedback in editing and writing of papers.

Acknowledgments

Above all, I thank Almighty God for strength and perseverance in times of weariness. I continue to progress because You have set great things for me.

I would like to thank my father, mother, brothers and their families who have patiently waited for me, motivated me, and above all strengthened me with their love and unrelenting support during the fulfillment of my goals.

Most importantly, sincere thanks and gratitude to my supervisor Dr. Nazim H. Madhavji and co-supervisor Dr. Mechelle Gittens. Without your support, guidance and tireless effort this thesis would not have been possible. I am truly indebted to your invaluable supervision and priceless advices that helped me to cultivate the skills to succeed in this endeavor and those advices will continue to influence me throughout my career.

I would like to acknowledge the support of all current and former colleagues in my lab, staff members (especially Graduate Secretary Janice Weirsma) and professors at Western, and other colleagues who have been instrumental in developing a congenial and conducive research environment. In particular, I am grateful to Zude Li for his suggestions on my thesis, Andriy Miranskyy of IBM for his help in data collection, Mark Wilding of IBM for providing me with resources needed to conduct an industrial-scale study at IBM, and Dr. Jamie Andrews for providing the mutation tool for this research.

I am also thankful to the University of Western Ontario for giving me the opportunity to conduct doctoral studies with full financial support during the last four years. I would also like to thank Ontario Graduate Scholarship for funding me during the course of this research and NSERC Canada for providing the funding necessary to collaborate with industry. I would also like to acknowledge IBM, Canada, for providing the resources needed for this research.

Thanks to all my friends who made my time in London memorable and thanks to my former classmate Bilal Ahmed for his suggestions on the F007 technique.

Table of Contents

List of Tables

List of Figures

Chapter 1

1 Introduction

According to an industrial poll (Erlikh, 2000), 85-90% of software systems' budget goes to software maintenance. Considering the high cost of software maintenance several researchers measure the cost associated with different categories of software maintenance. Schach et al. (2003) measures change logs and code modules of three software products (a commercial real-time product, a Linux kernel, and the GCC compiler) and finds that 53.4% to 56.7% of maintenance time is spent in corrective maintenance, 36.4-39% of time in perfective maintenance and 2.2-4% in adaptive maintenance. Lee & Jefferson (2005) also measures a Web-based Java (TM) application which reveals that 32% of maintenance time is spent in corrective maintenance along with 62% in perfective maintenance and 6% in adaptive maintenance.

The time spent in adaptive maintenance is small, and the time spent in perfective maintenance is difficult to avoid because it deals with the addition of new features or functionality. The time spent in corrective maintenance, however, is alarming because it deals with faults in a software system. Faults in a software system negatively affect software quality, customers' businesses and the market reputation of an organization. Thus corrective maintenance requires greater attention of the software engineering researchers.

Studies also show that in large software products about 50-90% (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) of the field failures are "rediscoveries" of the previous faults and 20% of the code is responsible for 80-100% of the faults (Boehm & Basili, 2001; Gittens et al., 2005; Ostrand et al., 2005). Despite knowing this, every field failure again requires the same effort in identification of the fault origin (Brodie et al., 2005; Lee and Iyer, 2000) and consumes more resources (again and again) on the same fault (Brodie et al., 2005; Lee and Iyer, 2000). According to our own discussions with developers of a large organization, identification of the fault origin can consume 30%-40% time of corrective maintenance (Proprietary Workshop, 2008). If faulty code (e.g., components,

functions) can be easily or automatically identified, then this will significantly reduce the corrective maintenance effort, cost and time.

1.1 Research Problem

This thesis, therefore, addresses the problem of identifying faulty functions in the execution traces[1] of crashing and non-crashing failures of deployed software.

Previous techniques focusing on the field failures: (i) require many passing and failing traces to identify the fault origin (e.g., using statistical debugging to identify important locations in the proximity of faulty source code (Chilimbi et al., 2009; Liu and Han, 2006)); (ii) discover only crashing failures by matching symptom of a problem with previously known faulty symptoms (Brodie et al., 2005; Lee and Iyer, 2000); (iii) identify faulty coarse grain code from execution traces such as files as the source of the field failure (Podgurski et al., 2003); and (iv) classify a field trace as passing trace or failing trace (Haran et al., 2007). Note that a non-crashing failure is more difficult to diagnose than a crashing failure because a non-crashing failure could manifest itself well after the executions of faulty code. However, the origin of the crashing failure can be readily identified by the same sequence of last function-calls after a crash.

1.2 Research Contribution

We solved the research problem, described in previous section, by developing an automated solution that identifies faulty functions in a (function-call level) failed trace of a deployed software system by using previously resolved (function-call level) failed traces of that program. Figure 1 shows a function-call level trace where “function entry” shows when control enters a function and “function exit” shows when it leaves a function. The proposed solution fills the gap of finer-grained discovery of fault origin (faulty function) from only a failed field trace of crashing and non-crashing failures.

---

[1] An execution trace is a record of the sequence of code labels (e.g., statements, functions) executed during a particular run of a program (IEEE Std. 610.12, 1990).

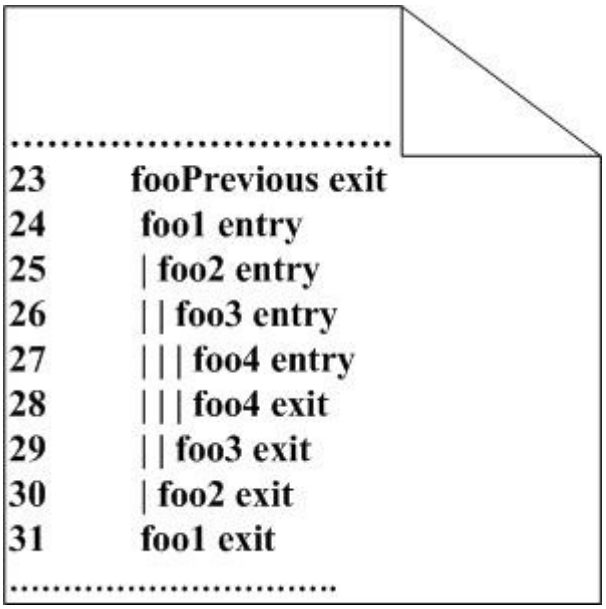


Figure 1: A function-call level execution trace.

The contributions of this thesis are:

(a) A technique F007 that identifies the list of suspected faulty functions in a trace of field failure by training the decision tree on previous failed traces of past releases and current release of a program (see Section 2.3). F007 can discover new and rediscovered faults in a function if traces of at least one (same or different) fault in that function are present in a previous collection of failed traces (see Section 2.6.2 and Section 2.7.6).

(b) A novel strategy called F007-plus that trains F007 on the failed traces of mutants[2] (artificial faults) and previous actual faults. F007-plus facilitates F007 to discover new faulty functions that were not discovered by F007 – F007 was able to discover only those faulty functions that were found previously faulty due to actual faults (see Chapter 3 and Section 3.5).

[2] Mutants are automatically seeded faults generated by changing statements of a program (Offutt et al., 2001).

(c) Identification of faulty functions in a trace of a current release by using the traces of prior faults in that function with the help of F007 and F007-plus. Traces of prior faults are obtained from pevious releases, mutants (artificial faults), or the current release (see Section 2.6.2 and Section 2.7; and Section 3.8).

(d) Faulty functions in approximately a maximum of 90% failed traces of subject programs can be identified on reviewing 10% or less of the code by using F007 and F007-plus (different settings result in different results; see, for example: Section 2.6.2 and Section 2.7; and Section 3.8).

(e) A discovery that only "function entry" or "function exits" in a function-call level trace (see Figure 1) are adequate to discover a fault's origin and their combine use does not affect the accuracy of identifying the fault's origin. This discovery reduces the size and run time overhead of the function-call level trace to approximately half (see Section 2.6.4).

(f) Patterns (or combinations) of function-calls (e.g., three functions that always appear together in a trace) do not produce better results than single function-calls when used with the decision tree algorithm (see Section 2.6.1).

(g) Faults generated using mutants (artificial faults) can be used to discover real faults (see Section 3.7 and Section 3.8).

(h) Traces (function-calls) of different faults in a group of related functions are similar; and traces of faults in one group of functions are different from traces of faults in another group of functions (see Section 3.7).

(i) An emerging theory: *"A faulty function can be so identified if the traces of at least one fault in that function are already known; and the accuracy of identification increases with the decreasing proportion of faulty functions in the program."* This emerging theory has been validated by using the criteria to measure goodness of theories by Sjøberg et al. (2008) in Section 4.4 of this thesis, and it is derived from two studies of this thesis, which are the core components of

this dissertation. The details of these two studies are described in the next section (see Chapter 4).

These research contributions are significant for deployed systems when only a few traces are available which is mostly the case because: (a) collection of multiple passing and failing traces for deployed systems can impede business operations by incurring extra overhead on them; and (b) collection of many traces consumes customers' resources and time. F007 (including F007-plus) addresses this need by using only the "failed" and reduced (i.e., "function entry" OR "function exit") function-call level traces. F007 is also very useful in the field testing such as alpha testing and beta testing. F007 is significant for the corrective maintenance from the point of view of reducing effort in locating the finer-grained fault origin (faulty functions) in the traces of the field failures.

1.3 Thesis Structure

The contributions of this thesis span the two studies of this dissertation. These two studies form the core of this thesis and are characterized in Table 1 with the following information: chapter number where the study is described in this thesis, study title, distinct characteristics of one study from another study, and publications year with the publications venues. Table 1 briefly provides the description of these studies by highlighting their differences.

The first study (in Chapter 2) proposes F007 and shows that: (a) patterns of function-calls are not better than single function-calls in discovering the fault origin; (b) only "function entry" or "function exit" are sufficient to discover the fault origin; and (c) different faults in the same function have similar function-calls. The second study, in Chapter 3, proposes F007-plus and shows that: (a) different faults in closely related functions occur with similar function-calls; and (b) mutants can be used to discover actual faults. Thus, it can be observed from Table 1 that these two studies are incrementally built over each other with the general focus on identifying faulty functions in the field traces.

These studies were conducted on twelve different programs and their many releases such as: (a) seven programs of the well known Siemens suite (Do et al., 2005); (b) the Space

program (Do et al., 2005); (c) Flex, Grep, Gzip and Sed--the four open source UNIX utilities--and their four to five releases (Do et al., 2009); and (d) a very large scale commercial program deployed in the field for about 20 or more years, with 20 million LOC and 200,000 functions. We evaluated F007 and improved strategy F007-plus by using a metric, also used by other researchers working on fault localization such as Jones and Harrold (2005) and Chilimbi et al. (2009), that quantifies a developer's effort in discovering fault locations. This effort measurement metric measures the number of functions or statements reviewed in discovering faulty functions or component. For example, F007 can identify faulty functions in 70-90% of the field failures on reviewing 10% or less of the program (i.e., 2-3 functions; see Section 2.6.2).

Table 1: Thesis core.

Chap. #	Study Title	Distinct Characteristics	Publications
2	F007: Finding Faulty Functions from the Function-call level Traces of the Field Failures	F007 is proposed to identify rediscovered faulty functions in the Siemens suite, the Space program, the UNIX utilities, and the large software system. Also identifies function-calls similarities amongst different faults in the same function.	ISSRE (Murtaza et al., 2008); CASCON(Murtaza et al., 2010); IEEE Trans. (submitted, Murtaza et al., 2011)
3	Using Mutants to Discover New and Rediscovered Field Faults by Exploiting the Similarity of Traces among Different Faulty Functions	F007-plus improves F007 by identifying new faulty functions using mutant (artificial faults) traces. Also identifies the similarity in function-calls of closely related faulty functions. The study was evaluated on the UNIX utilities and the Space program.	ICSE (Murtaza et al., 2011); IEEE Trans. (submitted, Murtaza et al., 2011)

This thesis is actually documented in the "integrated-article" format. Official guidelines pertaining to the "integrated-article" format can be found on the website of the University of Western Ontario[3]. In the "integrated-article" format, each chapter is a separate study

[3] http://grad.uwo.ca/current_students/trg_3.htm

and contains its own introduction, related work, procedure, evaluation and bibliography. However, the studies should not be disparate and the dissertation should show the logical relation amongst them. In our case, the research problem is the same but solutions are different: the second solution incrementally extends and overcomes the limitation of the first solution. In our case, the related work also overlaps in the two studies because of the same research problem. Thus, this thesis is structured as follows: Chapter 2 and Chapter 3 articulate two studies as shown in Table 1; Chapter 4 derives and validates an emerging theory based on the findings of the two studies; and Chapter 5 concludes this thesis with the directions to future work.

1.4 References

Boehm, B. & Basili V., R. "Software Defect Reduction Top 10 List", *Computer,* Vol. 34, No. 1, IEEE CS Press, Jan. 2001, pp. 135-137.

Brodie, M.; Sheng Ma; Lohman, G.; Mignet, L.; Modani, N.; Wilding, M.; Champlin, J.; and Sohn, P. "Quickly Finding Known Software Problems via Automated Symptom Matching." *Proc. 2nd Int'l Conf. on Autonomic Computing,* Seattle*, USA*, June 2005, pp. 101-110.

Chilimbi, T. M.; Liblit, B.; Mehra, K.; Nori, A. V.; and Vaswani, K; "HOLMES: Effective Statistical Debugging via Efficient Path Profiling". *Proc. 31st Intl. Conf. on Softw. Eng*., IEEE CS, Canada, May, 2009, pp. 34-44.

Do, H., Elbaum, S. G.; and Rothermel, G. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." *Empirical Softw. Eng.*, Vol. 10, Springer, Oct. 2005, pp. 405-435.

Erlikh, L."Leveraging Legacy System Dollars For E-Business," *IT Professional*, Vol.2, No.3, May-Jun, 2000, pp.17-23.

Gittens M.; Kim Y.; and Godwin D. "The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software." *Proc. 29th Int'l Computer Softw. and Appl. Conf.*, Edinburgh, Scotland, July 2005, pp. 179-185.

Haran, M.; Karr, A.; Last, M.; Orso, A.; Porter, A.A.; Sanil, A.; Fouche, S. "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks." *IEEE Trans. on Softw. Eng.* Vol. 33, No.5, May, 2007, pp. 287-304.

IEEE Std. 610.12, *Standard Glossary of Software Engineering Terminology*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

Jones, J. A. and Harrold, M. J., "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique". *Proc. 20th Int'l Conf. on Automated Softw. Eng.*, IEEE/ACM, CA, USA, 2005, pp.273-282.

Lee M. G. and Jefferson T. L. "An Empirical Study of Software Maintenance of a Web-based Java Application." *Proc. Int'l Conf. on Soft. Maintenance*, IEEE, Budapest, Hungary, Sep., 2005, pp. 571-576.

Lee, I. and Iyer, R. "Diagnosing Rediscovered Problems Using Symptoms." *IEEE Trans. on Sofw. Eng.*, Vol. 26, No. 2, Feb, 2000, pp.113-127.

Liu, C. and Han, J. "Failure Proximity: A Fault Localization-based Approach." *Proc. of the 14th SIGSOFT Symp. on Foundations of Softw. Engg.*, ACM, Portland, USA, Nov. 2006, pp. 45-56.

Murtaza, S.,S.; Gittens, M.; Li, Z.; Madhavji, N.,H.; "F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field". *Proc. Conf. of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ACM, Canada, Nov. 2010, pp. 61-75.

Murtaza, S.S.; Gittens, M.; and Madhavji, N.H. "Discovering the Fault Origin from Field Traces", Proc. of 19th *International Symposium on Software Reliability Engineering*, IEEE CS, Seattle, USA, Nov. 2008, pp. 295-296.

Murtaza, S.S.; Madhavji, N.H.; Gittens, M.; Li, Z.; "Diagnosing New Faults Using Mutants and Faults of Prior Releases (NIER Track)", Proc. of 33rd *International Conference on Software Engineering*, ACM, Honolulu, Hawaii, USA, May, 2011--*accepted.*

Murtaza, S.S.; Madhavji, N.H.; Gittens, M.; Li, Z.; Wilding, M.; Miranskyy, A.; Godwin, D.; "F007: Finding Faulty Functions from the Function-call level Traces of the Field Failures", *IEEE Transactions on Software Engineering,IEEE, USA,* 2011--*submitted.*

Murtaza, S.S.; Madhavji, N.H.; Gittens, M.; Li, Z.; "Using Mutants to Discover New and Rediscovered Field Faults by Exploiting the Similarity of Traces among Different Faulty Functions", *IEEE Transactions on Software Engineering*, IEEE, USA, 2011--*submitted.*

Offutt, A., J.; and Untch, R., H. "Mutation 2000: Uniting the Orthogonal," in *Mutation Testing for the New Century*, Wong W,E., Ed., USA: Kluwer Academic Publishers, 2001, pp. 34-44.

Ostrand T. J.; Weyuker E.; and Bell R. M. "Predicting the Location and Number of Faults in Large Software Systems." *IEEE Trans. on Software Engineering*, Vol. 31, No. 4, 2005, pp. 340-355.

Podgurski, A.; Leon, D.; Francis, P.; Masri,W.; Minch, M.; & Sun, J.; Wang, B, "Automated Support for Classifying Software Failure Reports". *Proc. Intl. Conf. on Software Engineering*, IEEE CS, Portland, US, May, 2003, pp. 465-475

Proprietary workshop on large commercial software, Sep., 2008.

Schach S. R.; Jin B.; Yu L.; Heller G. Z.; and Offutt J.; "Determining the Distribution of Maintenance Categories: Survey versus Measurement". *Empirical Softw. Eng.* Vol. 8, No. 4, Springer, Dec., 2003, pp. 351-365.

Sjøberg, D.; Dyba, D.; Anda, B. C.; and Hannay. J.; "Building Theories in Software Engineering", In *Guide to Advanced Empirical Software Engineering*. Shull,F., Singer,J., and Sjøberg,D.I.K, Eds., Springer, London, 2008, pp. 312–336.

Wood A. “Software Reliability from the Customer View.” *Computer*, Vol. 36, No. 8, IEEE CS, Aug., 2003, pp.37-42.

Chapter 2

2 F007: Finding Faulty Functions from the Function-call level Traces of the Field Failures

2.1 Introduction

Corrective software maintenance activity rectifies faults in a program (e.g., faults reported by users) (Chapin, 2000). Schach et al. (2003) measured change logs and code modules of three software products (a commercial real-time product, a Linux kernel, and the GCC compiler) and found that approximately 53% to 57% of the maintenance time is spent in corrective maintenance, approximately 36-39% in perfective maintenance and approximately 2-4% in adaptive maintenance. Also, Lee & Jefferson (2005) measured a Web-based Java (TM) application, revealing that 32% of the maintenance time is spent in corrective maintenance, 62% in perfective maintenance, and 6% in adaptive maintenance. Likewise, Sousa (1998) conducted a survey of large financial organizations in Portugal and reported that approximately 36% of the maintenance time is spent in corrective maintenance, approximately 49% in adaptive maintenance and approximately 14% in perfective maintenance. Previous research thus suggests that corrective maintenance effort is significant and, thus, research aimed at reducing this effort should be of high priority in software engineering.

Studies also show that in large software products approximately 50% (Wood, 2003), 70% (Lee and Iyer, 2000) and 50%-90% (Brodie et al., 2005) of the field failures are "rediscoveries" of previous faults. However, when a fault in software is reported by a user, it is not immediately apparent which part of software caused the failure -- even if it is a rediscovery (Lee and Iyer, 2000). It requires that faulty components, and ultimately the lines of code where the fault originates (Brodie et al., 2005; Lee and Iyer, 2000) are again identified. Such rediscoveries consume substantial resources; for example, Brodie et al. (2005) reported that one third of all the time spent by support staff at IBM is spent in diagnosing only rediscoveries. According to our own discussions with a developer of a large organization, fault identification, including rediscovered and new faults, can consume 30%-40% of corrective maintenance time (Proprietary Workshop, 2008).

Also, studies have reported that as much as 100% of the field faults originate in 10% of the code (Gittens et al., 2005), and 92% of the overall faults originate in 20% of the files (Ostrand et al., 2005) — this is the "80-20 Pareto rule" for software. Thus, if 50-90% of the field failures are rediscoveries of previous faults, and the faults originate from 20% of the code, then 20% of the code is causing 50-90% of the field failures. If the faulty code (e.g., components, functions or such abstractions) can be easily or automatically identified, then this should significantly reduce the maintenance effort, cost and time.

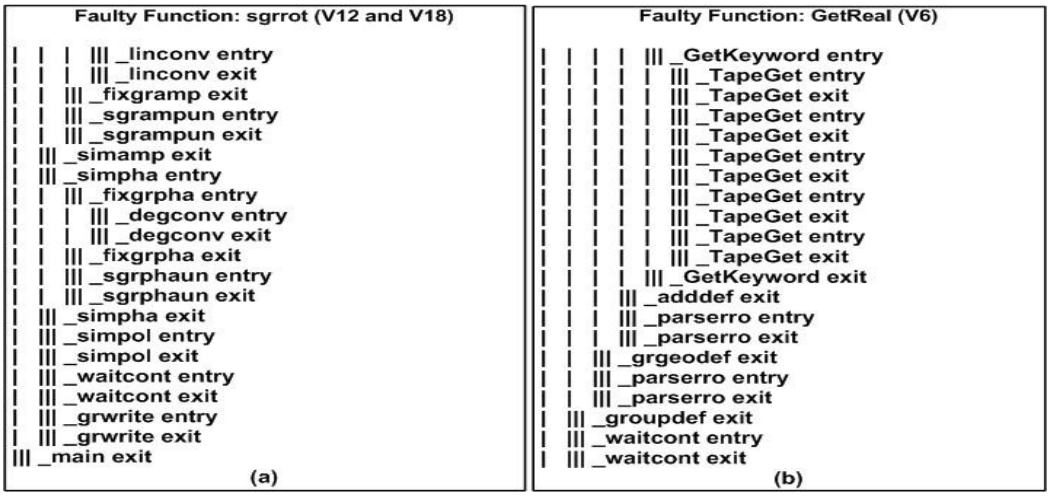


Figure 2: An example of common patterns in failed function-call level executions of the Space program.

During our preliminary experiments involving the Space[4] program (Do et al., 2005), we observed that if the same, or different, fault occurs in the same function then, in the most cases, the function-call level execution traces involving that function have similar function-call sequences. For example, Figure 2 (part 'a') shows that the sequence of last function-calls for two different faults (version[5] 12 and 18) in the same function "sgrrot" are exactly the same. On the other hand, in part 'b' the sequence of the last function-calls due to a fault in function "GetReal" is different from the sequence in part 'a'. Figure 2

[4] Space is a C program, an interpreter for an antenna array definition language written for the European space Agency and faults were found during development.

[5] In the Space program, each fault is equivalent to one version.

shows only the last function-calls but similar characteristics (with some variations) were also observed in the sequences of earlier function-calls. Moreover, in the cases of other faulty functions of the Space program, we observed similar characteristics as described for Figure 2.

These early observations, as explained above for Figure 2, warranted further empirical investigation. In particular, if these observations of similar function-call patterns held; and 50-90% of the faults were rediscoveries; and the Pareto rule also held, then the faults causing the majority of the field failures could be discovered by using the "failed" traces of a few earlier faults originating in a small percentage of the code. This is the fundamental motivation and basis of our investigation and it represents a divergent approach to locating faulty functions from execution traces.

This paper, therefore, addresses the problem of identifying the "function"-level origin of the field failures in the code. Our focus is on those field failures that can occur due to: (a) previously known faults or (b) a new fault in a previously known faulty function. The rationale behind this is rooted in empirical findings: (i) a significant amount (50-90%) of the field failures are rediscoveries (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003); and (ii) 80% of the field failures are concentrated in a relatively minor segment (20%) of a system (Gittens et al., 2005; Ostrand et al., 2005). Also, supporting this rationale is that faults in the same function occur with similar patterns of function calls (see Figure 2)[6]. In short, this paper focuses on using only a small part (20%) of the software system to solve the problem of identifying most (80-90%) of the rediscovered faults of a software system.

Our technique (called F007) uses a pre-classified, historical, collection of only "failed" function-call traces to identify faulty functions in newly collected failed traces. F007 trains a decision tree on the pre-classified collection to identify faulty functions in new failed traces. This pre-classified collection can be built from a collection of failed traces with the known faulty functions, obtained from the executions of software test cases. The

---

[6] Another example that can be drawn from practice is, when an out-of-bound index error occurs in an array in a function, then the sequence of the last function calls are usually the same.

reason for using in-house traces is in one of our studies on a very large commercial application (Gittens et al., 2005): we found an overlap in the location of faults in source code between field and pre-release failure; and we found that in general 20% of the code is responsible for 80% of the faults. Later on, failed traces from the field, as they get resolved, can be added to the collection, since the majority of faults are rediscoveries.

Previous techniques related to the field failures focus on: (a) classifying successful and failed traces (e.g., using decision trees (Haran et al., 2007), and using Markov model (Bowring et al., 2004)); (b) classifying rediscovered crashing failures via symptoms (Brodie et al., 2005; Lee and Iyer, 2000); (c) clustering traces related to coarse-grained code (e.g., files) (Podgurski et al., 2003); and (d) using statistical debugging (Chilimbi et al., 2009; Liu et al., 2005) to identify the origin of the field failures. The novelty of F007 can be readily seen from the contrast with previous approaches:

1. F007 can identify faulty functions (of rediscovered or new faults in the same function) in failed traces of both crashing (e.g., segmentation fault) and non-crashing (e.g., logical error) faults by using the knowledge of only previously resolved failed traces. In contrast, other techniques (Brodie et al., 2005; Lee and Iyer, 2000) identify faults in only crashing situations, identify faults at coarse grained level (e.g., files) (Podgurski et al., 2003), need knowledge of the type of a fault (Liu et al., 2005), and require a collection of passing traces with failing traces related to a fault (Chilimbi et al., 2009; Liu et al., 2005). It should be noted that: (a) the non-crashing failures are more difficult to solve because a user may not recognize the failure until well after the execution of the faulty code; whereas the origin of the crashing failure can be readily identified by the same last sequence of function-calls after a crash (Podgurski et al., 2003); and (b) collecting many passing and failing traces is not feasible in the field because of the overhead of trace collection and the user's resources consumption in fault reporting.

2. The techniques for "pass-fail" classification of field traces (Haran et al., 2007; Bowring et al., 2004) classify the traces as passing and failing. F007 builds upon

this by further examining only the failed traces to identify a faulty function in a failed trace.

3. When experimenting for F007, we discovered that only function entry OR exit points are required for finding the faulty functions in the failed trace. This reduces the overhead (runtime and size) by approximately half. This further distinguishes F007 from other techniques.

4. This paper further validates our previous work (Murtaza et al., 2010) on F007. Previously (Murtaza et al., 2010), we evaluated F007 only on the Siemens suite (Hutchins et al., 1994) which contains one release of seven small programs (128-494 LOC). This paper extends our work on F007: (a) by evaluating F007 on larger programs, such as: Space program (5767 LOC) (Do et al., 2005); several releases of four open source UNIX utilities. Flex, Grep, Gzip and Sed (4032-9831 LOC) (Do et al., 2005); and three releases of a very large commercial enterprise-level software of approximately 20 millions LOC and 200,000 functions; (b) by showing that F007 can use the traces of faulty functions of earlier releases to accurately identify the same faulty functions in subsequent releases; and (c) by showing that F007 can be used efficiently on very large programs by removing irrelevant function-calls. This further strengthens the significance of F007.

Our results of F007 on the Siemens suite, the Space program, four UNIX utilities, and the large software application show that: (a) F007 identified 75-90% of the faulty functions in the four UNIX utilities by reviewing less than 5% of the program, when trained only on 25% of the failed traces of the same release; (b) F007 discovered the same faulty functions (with different or the same faults) with 70% accuracy in the Siemens suite on reviewing 20% of the program when trained on 25% failed traces of at most one fault of the same faulty functions; (c) F007 identified faulty functions with 96% accuracy in the Space program on reviewing 5% of the code when trained on only 1% of the failed traces with at most one fault in the same function; and (d) F007 identified faulty functions in the subsequent release by using the failed traces of earlier releases and 10% failed traces of a current release with an accuracy of (i) 70-95% in the four UNIX utilities (that we studied)

on reviewing 5% or less of the code and (ii) 65-80% in the very large software system on reviewing 3% or less of the code. A direct comparison of F007 with other techniques does not exist (see Section 3.10). However, previous approaches that use function-call level traces (e.g., Di Fatta et al. (2006) and Tarantula (Jones and Harrold, 2005) on function coverage by Di Fatta et al. (2006)) identify only up to 40% faults (i.e., faulty functions) by reviewing 20% of the code in the Siemens suite; whereas, F007 can identify 70% faults (i.e., faulty functions) by reviewing 20% of the code.

These results imply that: (a) F007 can identify faulty functions in majority of the failed traces by using a small percentage of the previous failed traces; and (b) faults in the same function occur with similar function-call traces. F007 actually identifies faulty functions in a failed trace which makes it important when only a few traces are available for the fault discovery from deployed software—mostly the case due to trace overhead or user's resources consumption in fault reporting. F007 is therefore valuable since most of the field failures are rediscoveries of the faults originating from the same area of source of code. Thus F007 has the potential to reduce the corrective maintenance effort from the point of view of locating the finer-grained fault origin (faulty functions) from the field.

This paper continues as follows: Section 2.2 describes related work; Section 2.3 describes the F007 technique; Section 2.4 discusses the experimental setup; Section 2.5 shows an example of F007 from the Siemens suite (Do et al., 2005; Hutchins et al., 1994) ; Section 2.6 evaluates the results of executing F007 with the Siemens suite, the Space program (Do et al., 2005) and the UNIX utilities (Do et al., 2005); Section 2.7 validates F007 on a very large commercial application; Section 2.8 elaborates on the example execution from the perspective of prediction of faulty functions across releases; Section 2.9 summarizes the results on all the programs; Section 2.10 compares F007 against the contemporary techniques; Section 2.11 explains threats to validity; and Section 2.12 concludes and describes future work.

2.2 Related Work

Scientific literature describes a number of fault discovery techniques: (a) in-house fault localization techniques such as evaluating statement coverage (Jones and Harrold, 2005;

Wong et al., 2007) and recording function sequences (Dallmeier et al., 2005; Di Fatta et al., 2006); and (b) fault discovery techniques for field failures scuh as statistically identifying field failures (Liu and Han, 2006; Podgurski et al., 2003), techniques for classifying field failures (Bowring et al., 2004; Haran et al., 2007; Elbaum et al., 2007), techniques for rediscovering known problems (Brodie et al., 2005; Lee and Iyer, 2000), and techniques for localizing system configuration level faults (Chen et al., 2004; Ding et al., 2008; Yuan et al., 2006). In the following sub-sections, we elaborate on each of these techniques.

2.2.1 In-house Fault Localization Techniques

Agrawal et al. (1995) present a heuristic method which considers that the fault lies in the difference between a successful test case execution and a failed test case execution, known as a dice. Wong et al. (2006) present two improvements over this dice-based technique (Agrawal et al., 1995): (a) if a bug is not found in a dice, then the first improvement, called augmentation, incrementally includes the code from the intersection of a failed test case and a successful test case; and (b) if a dice is too big then another improvement, called refining, gradually decreases the code by differentiating the dice with another successful test case

Jones and Harrold (2005) and Wong et al. (2007) discover faulty statements in a program by using a theory that statements executed by passing test cases are less likely to be faulty compared to those executed by failing test cases. Zhang et al. (2009) identify suspected statements by contrasting edge profiles (i.e., paths executed when control moves from one branch to another) of failing and passing runs. They (Zhang et al., 2009) argue that propagating faulty program states can be captured using edge profiles, which can identify faulty statements better than coverage based techniques such as the technique by Jones and Harrold (2005).

Di Fatta et al. (2006) and Dallmeier et al. (2005) propose a technique that localizes the faulty functions and the faulty classes, respectively, by extracting function-call patterns from the failed and successful executions. Statistical debugging based techniques (e.g., Zheng et al. (2009) and (SOBER) Liu et al. (2005)) instrument source code with light

weight assertions (e.g., null pointer check) and then apply a statistical utility function on the assertions obtained from failed and successful executions. This is to discover faulty assertions (e.g., null pointers).

These techniques are suitable for in-house testing but not for deployed software because: (a) mostly they require a collection of failing traces related to one fault[7], but failure traces in the field do not necessarily result due to the same fault; (b) in deployed software, often only a few traces (at the time of fault) are available due to the overhead incurred in trace collection, and sometimes it is not known if a trace is passing or failing; (c) different customer usages can result into many different normal execution paths that are not observed in passing executions in in-house testing, and it is not feasible to collect many passing traces from customers due to overhead involved in trace collection; (d) finer grained (e.g., statement coverage) coverage profiles are costlier to collect than function coverage; and (e) the statistical-debugging based techniques require the knowledge of the type of bug for instrumentation, and if a fault is not found another type of assertion (e.g., null pointer check) is instrumented in source code, and so on the process is repeated until the fault is found.

2.2.2 Statistically Identifying Field Failures

Podgurski et al. (2003) form clusters of execution traces of the field failures based on common faulty source files. The granularity in the Podgurski et al. approach is a faulty file, whereas the majority of the clusters contained failed traces with multiple files (fault origin), making it unsuitable for the manual investigation of the correct faulty file (and finer-grain origin gets even more difficult). In contrast, F007 discovers faults automatically at the finer-grained, function-level; and the faults in the majority of traces can be discovered correctly by reviewing the first suspected function. Podgurski et al. (2003) experimented on GCC, Javac and Jikes; whereas, we experimented on the Siemens suite, the Space, the UNIX utilities and the large commercial program.

---

[7] One of the Dallmeier et al. (2005) methods uses one failed trace and a collection of passing traces.

Liu and Han (2006) cluster failing runs according to a rank list of assertions obtained using the statistical debugging tool SOBER (Liu et al., 2005). In the approach by Liu and Han (2006), the fault locations (in the Siemens suite) are discovered by following the top rank predicates in the list produced using SOBER. However, their approach also requires a collection of the passing traces and the failing traces for the same fault. Their work (Liu and Han, 2006) also suffers from the limitations of statistical debugging (see Section 2.2.1). F007 can discover faulty functions without such limitations.

Another statistical debugging tool, HOLMES (Chilimbi et al., 2009), uses path profiles to classify faults for deployed software. Their technique can only be applied to the server side applications (Chilimbi et al., 2009), because they have to redeploy software components with instrumentation of selected functions to collect the passing traces and the failing traces pertaining to one fault (a limitation, see Section 2.2.1). In some cases, this may not be feasible for running servers as well due to the runtime redeployment of instrumented software components.

2.2.3 Classifying "Pass-fail" Field Traces

Elbaum et al. (2007) experiment with different anomaly detection algorithms that detect abnormal behavior in deployed system. They identified abnormal behavior by comparing the deployed system's (function-call level) trace with the model of in-house passing traces. Their objective is to anticipate the occurrence of a failure such that trace collection for the failure could be automatically started at the right time when the system is in the field. Bowring et al. (2004) and Haran et al. (2007) develop techniques based on the Markov model (Bowring et al., 2004) and decision tree (Haran et al., 2007) to characterize (statement, branch or function level) executions as being passed or failed runs. These techniques complement our work in discovering the faulty functions in that the F007 technique requires only failed traces. For example, if traces collected from the field also contain some of the passing traces, they can be filtered out by using the techniques proposed by Haran et al. (2007) and Bowring et al. (2004). The failed traces can be collected directly from the field using Elbaum's et al. (2007) technique. Subsequently, F007 could use the filtered (failed) traces to localize faults.

2.2.4 Rediscovery of Problems

Brodie et al. (2004) use string matching to group one function-call trace of a crash with other groups of function-call traces for different crashes. The groups of crashes were formed by exactly matching the function-call paths of different crashes. They claim that every group, formed on the basis of the same trace matches, has the same crashing reason. However, traces due to the same crashing reason (or same fault) are not exactly the same, and they can take different approaches. Lee and Iyer (2000) propose a technique to classify the rediscovered crashing failure by literal matching of its function-call trace with already known failure traces. They consider a variety of heuristics to match several function-call paths followed by the same fault. In F007, we model several paths leading to the same fault by the decision tree algorithm. F007 addresses the more difficult problem of non-crashing failure classification, where a user may notice a failure well after the execution of the faulty code (Podgurski et al., 2003). For example, if a Statement object is not de-initialized in a Java program, memory consumption continues to increase, and application slows down due to swapping by OS: this occurs long after many functions execute that object. Similarly, initialization of a wrong value to a variable can influence in the wrong output well after the manipulation of that variable. F007 can discover faulty functions in both the non-crashing and crashing failure traces.

2.2.5 Localizing System Level Faults

Yuan et al. (2006) employ support vector machines (a classification algorithm) to determine the root causes of a problem (e.g., network cable unplugged) confronting a user of a system on the basis of execution traces of software. Chen et al. (2004) describe a technique based on the decision tree and association rule to diagnose faulty components (e.g., a web server) in large distributed systems. Ding et al. (2008) also propose a technique to identify faults occurring due to configuration of software. They collect software runtime behavior (e.g., system calls, environment variables) of the passing and the failing runs. The origin of the fault (e.g., large log files) is identified by deviations in failing and passing run. In summary, these three techniques also complement our work in that they focus on the identification of faults in application interactions or at system level; whereas, F007 focuses on the faults within an application. For example, systems level

techniques (Chen et al., 2004; Ding et al., 2008; Yuan et al., 2006) could first identify faults at the system level (e.g., memory overload); F007 can then identify logical faults within a program (e.g., infinite loop).

2.2.6 Research Gap

Table 2 characterizes closely related techniques in four categories: focus of the research; fault localization at a level of single trace or at a level of collection of traces for one fault; fault localization granularity; and the specific type of knowledge required to discover faults. Each category further classifies each technique. For example, the focus of research classifies each technique into testing and corrective maintenance.

Table 2. Characterization of F007 and closely related techniques.

- **Focus**: Testing (T); and Corrective Maintenance (CM).
- **Trace Contents:** Statement coverage (St); Function Sequence (FS); Predicate Count (PC); and Path Profile (PP).
- **Input**: Single Trace (Tr); and Collection of Passing Traces and Failing Traces for one Fault of a Program (PFV)
- **Fault Location Granularity**: Function (Fn); Statement (St), File (Fi), Predicates (check points in code)(P); Paths (Ph); Cause of Only Crashing Failures (hardware, software) (Cr); Execution Dice (ED); and Class (Cl).
- **Weakness**: Type of bug (TB); Collection of Passing Traces and Failing Traces for one fault of a program (PFV); and Historical Traces for Faults (HT).

Reference	Focus	Trace Contents	Input	Fault Location Granularity	Weakness
Agrawal et al. (1995)	T	St	PFV	ED	PFV
Wong et al. (2006)	T	St	PFV	ED	PFV
Wong et al. (2007)	T	St	PFV	St	PFV
Jones et al. (2005)	T	St	PFV	St	PFV
Di Fatta et al. (2006)	T	FS	PFV	Fn	PFV
Dallmeier et al. (2005)	T	FS	PFV	Cl	PFV
Liu et al. (2005)	T	PC	PFV	P	PFV, TB
Zheng et al. (2003)	T	PC	PFV	P	PFV,TB
Chilimbi et al. (2009)	CM	PP	PFV	Ph	PFV
Liu et al. (2006)	CM	PC	PFV	P	PFV, TB
Podgurski et al. (2003)	CM	FS	Tr	Fi	HT
Lee & Iyer (2000)	CM	FS	Tr	Cr	HT
Brodie et al. (2005)	CM	FS	Tr	Cr	HT
F007	CM	FS	Tr	Fn	HT

The closest techniques to F007 (i.e., techniques for corrective maintenance) focus on: (a) finding a fault in a trace at coarser (file) level (manual investigation of the finer-grain origin of fault remains difficult) (Podgurski et al., 2003); (b) discovering only crashing faults (Brodie et al., 2005; Lee and Iyer, 2000); and (c) identifying fault by using many passing traces and failing traces for a fault (Chilimbi et al., 2009; Liu and Han, 2006). F007 fills the gap of finer-grained discovery of fault origin (faulty function) from only a failed field trace of the deployed instance of a software application (by using previous failed traces from field or in-house testing). F007 distinguishes from these techniques in the following ways:

- F007 contributes by showing that different faults in the same function occur with similar function-calls (see Section 2.6.2).

- F007 can discover (new or old faults in the same) faulty functions in a failed field trace by reviewing only a small percentage of the code (e.g., first two to three functions; described in detail in Section 2.6.2 and Section 2.7).

- F007 shows that only "function-entry" or only "function-exit" are sufficient to identify faulty functions (or fault origin) from a function-call trace (see Section 2.6.4). This discovery facilitates in reducing the size and trace collection overhead to approximately half.

These distinguishing factors of F007 are significant for the deployed systems when only a few traces are available—mostly the case due to user's time consumption and trace overhead. Also, if most of the field failures are (50-90%) rediscoveries and originate from the same area (20%) of code, then (considering the above factors) F007 can discover the origin (faulty functions) of the majority of faults (see Section 2.6.2 and Section 2.7).

2.3 The F007 Technique

Recall from Section 2.1 that F007 identifies faulty functions in function-call level traces of field failures. F007 requires a historical collection of traces (or their reproductions)

from failed executions (e.g., as in Figure 2). F007 also assumes that the faulty functions have previously been identified in a historical collection of traces.

Also recall from Section 2.1 that an initial collection of traces can be built using in-house test case traces of software because there is an overlap between the origin of in-house and the field faults (Gittens et al., 2005). Subsequently, failed traces from the field can be added to the collection. This is because most of the faults (80-90%) are rediscoveries originating with similar patterns from the same area (20%) of source code.

Following are the primary two steps of F007:

Step 1: F007 first extracts patterns (combinations) of function-calls from the given function-call traces using the MINEPI algorithm (Mannila et al., 1997). (Function-call level execution traces are shown in Figure 2, where "function entry" and "function exit" show when control enters and exits a function respectively.) The MINEPI algorithm is used to identify temporal patterns in sequences; e.g., if function "TapeGet" is called in part 'b' of Figure 2 then another "TapeGet" would follow within the distance of one function-call.

Step 2: F007 then trains decision trees (Witten and Frank, 2005) on these patterns (obtained from historical traces of in-house or field faults) to discover faulty functions in a given set of new trace of failure. The decision tree is a well known classification algorithm (Witten and Frank, 2005). For example, F007 trains a decision tree on the historical collection of traces for each faulty function. Whenever a new failed trace arrives, each decision tree tries to associate the patterns in the new trace with its knowledge of patterns. Each decision tree then predicts its faulty function for a new trace with a probability. Predicted faulty functions are then arranged in a list in the decreasing order of their predicted probabilities. The list of faulty functions is then presented to developers with the knowledge that functions with the highest probabilities should be considered highly suspected faulty functions.

The intuition behind this approach to discovering patterns in traces is that function-calls related to similar faults constitute similar patterns (also noted in (Lee and Iyer, 2000)),

and the decision tree algorithm (in the second step) can further leverage these similar patterns by associating them to common faulty functions. An example of similar patterns of function-calls is shown in Section 2.1.

Section 2.3.1 describes step 1 of F007 (the MINEPI algorithm to extract patterns of function-calls), and Section 2.3.2 describes step 2 of F007 (the decision tree algorithm).

2.3.1 MINEPI

Mannila et al. (1997) propose two algorithms (WINEPI and MINEPI) to discover temporal patterns in log files such as telecommunication alarms and data logs for Web servers. The sequences of function-calls (see Figure 2) bear similar temporal relationships to each other. Association rules (Witten and Frank, 2005) do not discover such temporal relationships. For example, the time duration of two function-calls is not considered in the association rules; whereas the function-calls in a trace bear temporal relationships and MINEPI can extract the temporal relationship. MINEPI is preferred over WINEPI because it can discover patterns not discovered by WINEPI (Mannila et al., 1997).

Fundamental Definitions: We shall now summarize the MINEPI algorithm (Mannila et al., 1997) by using the notations and definitions used by Mannila et al. (1997), with minor modifications and simplifications:

- First, consider a set R of event types.

- An event is expressed as a pair (A, t), where:

  $A \in R$ is an event type, and t is an occurrence time of the event expressed as an integer --indicating the order of this event in the temporal sequence.

For example, in our case the name of a function in a function-call level trace is an event type (A) and the t is the calling order of the function in a function-call trace.

- An event sequence S on R can be formally expressed as a triple (s, Ts, Te), where:

Ts <= Te and Ts is the (integer) starting time, and Te is the (integer) ending time, and s= < (A_1, t_1), (A_2, t_2)……. (A_n, t_n) > is an ordered sequence of events such that Ai ∈ R for all i=1….. n, and $t_i <= t_{i+1}$ for all i=1….. n-1, and $T_s <= t_i < = T_e$ for all i = 1 to n.

For example, a sequence of events is shown in Figure 3 (which we use in examples in the rest of this section; serial and parallel episodes in Figure 3 are explained later). In Figure 3, foo_1, foo_2, foo_3 and foo_4 are the event types (functions), (foo1,1) is an example of the event (A,t), Ts= 1 and Te=12; and s= < (foo_1,1), (foo_2,2)…… (foo_4,12) >.

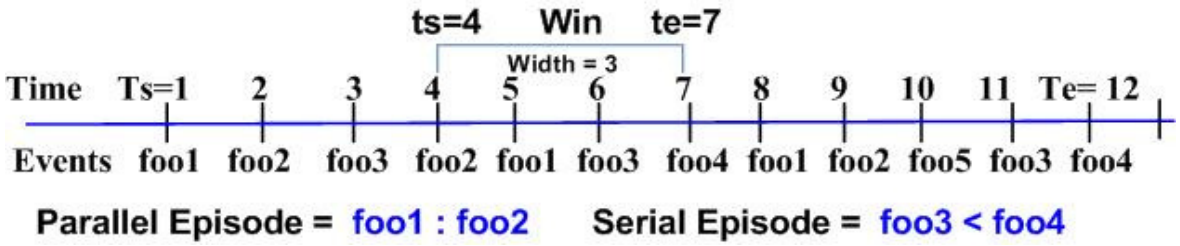


Figure 3: Event (function) sequences and episodes.

• An **episode** α is a pair (V, ≤) where V⊂ R and "≤" is a partial order relationship on V. If a partial order ≤ among predicates of an episode is a *trivial* partial order (unordered) then the episode is called a **parallel episode**; that is, $A_i \not\leq A_j$ for all A_i, A_j ∈ V such that $A_i \neq A_j$. If a partial order "≤" among predicates of an episode is a *total* (fixed) order then the episode is called a **serial episode**; that is, $A_i \leq A_j$ or $A_j \leq A_i$ for all A_i, A_j ∈ V. An episode α = (V, ≤) occurs in an event sequence S if: (i) for all A_i ∈ V there is an event (A_i, t_i) in S; and (ii) for all A_i, A_j ∈ V with $A_i \leq A_j$ and $A_i \neq A_j$ there are events (A_i , t_i) and (A_j, t_j) in S such that $t_i < t_j$. The **length 'L'** of an episode α is the number of event types in an episode, i.e., |V|.

In this text, we **denote trivial partial order by ":" symbol** between two event types of a parallel episode and **a total (fixed) order by "<" symbol** between two event types of a serial episode. Suppose A_1, A_2 …. A_n ∈ V then we represent a serial episode as "$A_1 < A_2$ ….< A_n" and a parallel episode as "A_1 : A_2 ….. : A_n". For example, "$foo_3 < foo_4$"

represents a serial episode of Length 2 in Figure 3, and it shows that foo_3 precedes foo_4. Similarly, "foo_1 : foo_2" is a parallel episode of Length 2 in Figure 3 that shows that foo_1 and foo_2 occcur together but not in a fixed order.

- A episode $\beta = (V', \leq')$ is a **sub-episode** of $\alpha = (V, \leq)$, denoted by $\beta \subset \alpha$, if there exists an injective mapping f: $V' \rightarrow V$ such that $f(A') = A$ for all $A' \in V'$ and $A \in V$, and for all $A_i', A_j' \in V'$ with $A_i' \leq' A_j'$ then $f(A_i') \leq f(A_j')$ such that $f(A_i') = A_i$ and $f(A_j') = A_j$ and $A_i, A_j \in V$.

- A ***window*** on an event sequence S (s,Ts,Te) is also an event sequence W (w, ts, te), where: ts <= Te, te >= Ts, ts <= te, and w consists of the events (A_i, t_i) with $t_s <= t_i <= t_e$ and **Window width, win(w)**, is the time span te –ts.

An example of the window width of win (w)= 3 is shown in Figure 3, and it is W (<(foo_2, 4), (foo_1,5), (foo_3,6), (foo_4,7) >, 4, 7).

- The **minimal occurrence** of an episode α in an event sequence S is the interval [ts, te], if α occurs in a window W(w, ts, te) and α does not occur in any proper sub-window W'(w', ts', te') such that ts<= ts' and te'<=te and win(w') < win(w). The set of minimal occurrences of an episode α in a given event sequence is denoted by $mo_{win\,(w)}(\alpha)$:

$$mo_{win\,(w)}(\alpha) = \{\, [ts,te] \mid [ts,te] \text{ is a minimal occurrence of } \alpha \,\}$$

- The number of minimal occurrences of an episode in a sequence is the ***frequency***[8] "$|mo_{win(w)}(\alpha)|$" of an episode.

Thus, we extract only those episodes which have minimal occurrences for a window width. For example:

---

[8] Mannila et al. (1997) used the term "support" ; for simplicity, we use he term "frequency" in this paper

(1) Serial episode "$foo_1 < foo_2$" minimally occurs twice ($|mo_4(foo_1<foo_2)|=2$) in the event sequence in Figure 3 during the intervals {[1,2],[8,11]} for the window width win(w)=4. In the interval [1,4] "$foo_1<foo_2$" is minimally true for "foo_1" at t_1 and "foo_2" at t_2, but not for 'foo_2' at t_4 because there exists 'foo_2' at t_2 which is true for the minimal occurrence of "$foo_1 < foo_2$" in the interval [1,4]. Length 'L' of the episode "$foo_1< foo_2$" is 2.

(2) Parallel episode "$foo_1 : foo_2$" (length L=2) minimally occurs in the event sequence of Figure 3 for the win(w)=2 at the intervals [1,2], [4,5], [8,9].

(3)Parallel episode "$foo_1:foo_2:foo_3$" (Length L=3) is minimally true for win(w)=3 in intervals {[1,3], [3,5], [4,6], [6,9],[8,11]}, and is not minimally true for the interval [2,5] because there exists a sub-interval [3,5], i.e., [3,5] is the minimal occurrence within the interval [2,5]. Similarly, it is also not minimally true for the interval [1,4] because [1,3] is the minimal occurrence.

- An **episode rule** is an expression of the form β [*win*(w_1)] $\Rightarrow$ α [*win*(w_2)], where $\beta \subset \alpha$ (β is a sub-episode of α) and *win*(w_1), *win*(w_2) are integers. Episode β has a minimal occurrence at the interval [*ts,te*] with window $W_1(w_1, ts, te)$ and $te - ts \leq win(w_1)$, and episode α has the minimal occurrence at the interval [ts,*t'e*] with window $W_2(w_2, ts, t'e)$ for some *t'e* such that $t'e - ts \leq win(w_2)$ and $t'e > te$. **Confidence** of the rule β [*win*(w_1)] $\Rightarrow$ α [*win*(w_2)] is given as:

$$\frac{|\{mo_{win(w_1)}(\beta) \mid mo_{win(w_2)}(\alpha)\}|}{|mo_{win(w_1)}(\beta)|}$$

For example, "foo1" is the only sub-episode of the serial episode "foo1 < foo2" that could be extracted according to the definition of episode rule (i.e, a subepisode should begin at the same time as an episode and subepisode's time interval should not be greater than an episode). The episode rule for this serial episode with *win*(w_2)=4 and *win*(w_1)=3 is "foo_1[3] => (foo_1< foo_2)[4]". Confidence of this episode rule, from the event sequence in Figure 3, is shown below:

$$\frac{|\{mo_3(foo_1)\,|\,mo_4(foo_1 < foo_2)\}|}{mo_3(foo_1)} = \frac{2}{3}$$

This rule is read as "if 'foo_1' minimally occurs within the window width of 3 time units, then there is a 67% (2/3=0.67) chance that 'foo_2' will follow 'foo_1' such that "$foo_1 < foo_2$" occurs minimally within 4 time units."

Executing MINEPI: The MINEPI algorithm starts by first extracting minimal occurrences of all the episodes of length L=1. Subsequently, it extracts minimal occurrences of all the higher length episodes (serial or parallel) of length L = 2, 3 to n incrementally for a particular window width. Maximum value of the length 'L' and the window-width is decided by the user in the MINEPI algorithm. Also, the minimum frequency is set by the user. This is to select the frequent episodes beyond a minimum frequency value. For example, a user can select episodes with minimum frequency greater than 5.

After extracting all the episodes, episode rules are generated for each episode by identifying ***every*** **sub-episode of an episode** as per the definition of episode-rule. An episode rule $\beta[win(w_1)] \Rightarrow \alpha[win(w_2)]$ is generated if conf ($\beta[win(w_1)] \Rightarrow \alpha[win(w_2)]$) is greater than the minimum confidence. (The minimum confidence for an episode rule is also set by a user.) To keep things simple: we chose $win(w_2)$ in the rule to be the maximum window width set by the user and set $win(w_1) = win(w_2) - 1$, for all our rules; and we also write the rule $\beta[win(w_1)] \Rightarrow \alpha[win(w_2)]$ as $\beta \Rightarrow \alpha$. For example "$foo_1[3] \Rightarrow (foo_1 < foo_2)[4]$" can be written as "$foo_1 \Rightarrow foo_1 < foo_2$" that is foo_1 within window width 3 precedes foo_2 with 67% chance within a window width of 4.

It should be noted that with larger values of window width and episode length, a large number of episode rules will result. On the contrary, smaller values of minimum frequency and minimum confidence can also result in a large number of episode rules, which would be infeasible for processing and memory utilization. However, we must not miss important episode rules necessary for classification. Therefore, we set minimum frequency to greate than zero (i.e., 1) and minimum confidence to greater than zero, and to keep overhead minimum we decided to vary episode length and window width. This was done until we found the best combination of episode length and window width, that

is, the one which can predict the faulty function (using decision trees) with the highest accuracy.

Execution Example: Suppose, we have extracted a parallel episode "foo_1:foo_2:foo_3" of length 3 for win(w_2)=4 as set by a user. First we identify all the sub-episodes; that is, sub-episodes of length 1 {foo1, foo_2, foo_3}, and the sub-episodes of length 2 {foo_1:foo_2, foo_2:foo_3, foo_1:foo_3}. Second, we generate an episode rule for every sub-episode β *and* episode α. So, six episode rules can be generated in all for this parallel episode corresponding to each of the six sub-episodes with minimum confidence and frequency greater than 0. This is shown in Table 3 where: first column shows sub-episode, second column shows the episode rule using sub-episode and episode, third column shows how confidence is calculated, fourth column shows confidence value using Figure 3, and fifth column shows an informal interpretation of the rule.

Table 3: Epsiode rules from a parallel episode of length 3.

α=foo_1:foo_2:foo_3 and win(w_2)=4 and win(w_1)=3				
β	**β=>α**	**Confidence**	**Conf. Val**	**Informal Interpretation (if $mo_3(\beta)$=true and $mo_4(\alpha)$=true)**
foo_1	foo_1 => foo_1:foo_2:foo_3	$\lvert\{mo_3(foo_1) \mid mo_4(foo_1 : foo_2 : f\ oo_3)\}\rvert$ / $\lvert\{mo_3(foo_1)\}\rvert$	2/3	foo_1< (foo_2:foo_3)
foo_2	foo_2 => foo_1:foo_2:foo_3	$\lvert\{mo_3(foo_2) \mid mo_4(foo_1 : foo_2 : f\ oo_3)\}\rvert$ / $\lvert\{mo_3(foo_2)\}\rvert$	1/3	foo_2 < (foo_1:foo_3)
foo_3	foo_3 => foo_1:foo_2:foo_3	$\lvert\{mo_3(foo_3) \mid mo_4(foo_1 : foo_2 : f\ oo_3)\}\rvert$ / $\lvert\{mo_3(foo_3)\}\rvert$	2/3	foo_3 < (foo_1:foo_2)
foo_1 : foo_2	foo_1:foo_2 => foo_1:foo_2:foo_3	$\lvert\{mo_3(foo_1:foo_2) \mid mo_4(foo_1 : foo_2 : f\ oo_3)\}\rvert$ / $\lvert\{mo_3(foo_1:foo_2)\}\rvert$	3/3	(foo_1:foo_2) < foo_3
foo_1 : foo_3	foo_1:foo_3 => foo_1:foo_2:foo_3	$\lvert\{mo_3(foo_1:foo_3) \mid mo_4(foo_1 : foo_2 : f\ oo_3)\}\rvert$ / $\lvert\{mo_3(foo_1:foo_3)\}\rvert$	1/5	(foo_1:foo_3) < foo_2
foo_2 : foo_3	foo_2:foo_3 => foo_1:foo_2:foo_3	$\lvert\{mo_3(foo_2:foo_3) \mid mo_4(foo_1 : foo_2 : f\ oo_3)\}\rvert$ / $\lvert\{mo_3(foo_2:foo_3)\}\rvert$	1/5	(foo_2:foo_3)< foo_1

For example, in the first row of Table 3 the rule "foo_1=> foo_1: foo_2 : foo_3" is read as if "foo_1 " occurs minimally in 3 time units then there is a (2/3) 67% chance that "foo_2 : foo_3" will follow such that "foo_1: foo_2 : foo_3" occurs minimally in 4 time units. Informally, this can be interpreted as "foo_1" minimally occurs within 3 time units in a total-order with unordered "foo_2 and foo_3" such that "foo_1: foo_2 : foo_3" occurs minimally

in 4 time units; that is, "$foo_1 < (foo_2 : foo_3)$" given that $mo_3(foo_1)$=true and $mo_4(foo_1 : foo_2 : f\,oo_3)$=true (recall that "<" denotes total order and ":" denotes no-order).

Similarly, consider another episode rule "$foo_1:foo_3 \Rightarrow foo_1:foo_2:foo_3$" and it is read as if "$foo_1:foo_3$" occurs minimally in 3 time units then there is 20% chance that "foo_2" will follow such that "$foo_1:foo_2:foo_3$" minimally occurs in 4 time units. An informal intepretation "$(foo_1:foo_3) < foo_2$" is also shown in Table 3. In other words, parallel episode rules are the combination of serial and parallel episodes – i.e, they are actually hybrid episodes. Finally, an example of creation of episode rules from serial episodes is already shown above.

Confidence of Length 1 Episodes: In the MINEPI algorithm, by definition the confidence of episodes of length 1 is always 1. We have modified this definition for the episodes of length L=1, such that:

$$Conf\ (\alpha_j) = |\, mo_0(\alpha_j)\,| \Big/ \sum_{i=1}^{n} |\, mo_0(\alpha_i)\,|$$

Where $A_1, A_2 \ldots A_n$ be the event types in a given event sequence, and α_i be the episode $(\{A_i\},\{\})$ with no partial order, and $mo_0(\alpha_i)$ is the minimal occurrence of α_i when win(w)=0 (i.e., when there is only single event).

It measures the chances of occurrence of length 1 episodes in an event sequence. For example, the confidence of foo_1 in Figure 3 is: Conf $(foo_1) = |\, mo_0\,(foo_1)| \,/\, \sum |\, mo_0\,(foo_i)|$ = 3/12 (where, i=1-4).

Conclusion: In short, the MINEPI algorithm extracts all kinds of temporal patterns (i.e., serial, parallel and hybrid) of different lengths and different window widths from a sequence.

2.3.2 Decision Tree Algorithm

Recall (from Section 2.3, lead text) that decision trees were trained on the episode rules (patterns of function-calls) to predict faulty functions in the failed traces. Figure 4 gives

an example of length 3 episode rules and a trace. A row represents a trace from a historical collection of failure traces, and cells represent the confidence of episode rules in a trace respectively. For example, first cell shows that an episode rule "foo1 => (foo1 < foo2 < foo3" has a confidence of 1 in the trace T1. The last column shows the faulty functions for a historical trace, already identified in a trace from history (as discussed earlier in Section 2.3, lead text). In data mining terminology, faulty function is a dependent variable and the episode rules of function calls are independent variables.

F007 employs one-against-all (Witten and Frank, 2005) approach in training the decision tree classifier. In this approach, a dataset (of traces) with M categories of dependent variable (faulty functions) is decomposed into M new datasets with binary categories. Each new binary dataset 'Di' has category 'Ci' (where i = 1 to M) labeled as positive and all other categories are labeled as negative. On each new datasets 'Di' the decision tree algorithm is trained; resulting in 'M' trees in total.

	Episode Rules						
Traces	foo1 => (foo1 < foo2 < foo3)	(foo1 < foo2) => (foo1 < foo2 < foo3)	(foo2 < foo3) => (foo2 < foo3 < foo1)	foo3 => (foo3 < foo4 < foo1)	.....	foo5 < foo6 => (foo5 < foo6 < foo1)	Faulty Functions
T1	1.0	0.67	0.48	0.10		0.21	foo3

Figure 4: Length 3 serial episode rules and a trace with confidence and pre-known faulty functions from history.

Whenever a new faulty trace arrives, each decision tree predicts its category 'Ci' of the dependent variable (faulty function) along with a probability of being faulty. Predicted faulty functions are then arranged in a list in the decreasing order of their predicted probabilities. Empirical evidence (Polat and Güneş, 2009) shows that training multiple

decision trees (one-against-all) on several binary datasets yields better results than training a single decision tree on a dataset with many categories of dependent variable.

In fact, we employed the one-against-all approach with a little modification; i.e., instead of selecting the predicted faulty function with the highest probability, we ranked the predicted faulty functions in the decreasing order of their predicted probabilities. The reason is that: (a) a developer gets multiple options if the function with the highest probability is not the actual faulty function, (b) a developer's effort could be quantified using a metric to estimate effort in discovering the fault (Jones and Harrold, 2005; (Di Fatta et al., 2006) (e.g., percentage of code reviewed in discovering the fault), and (c) using the developer's metric, the comparison against other techniques gets simpler. The function list is then presented to a developer with an intuition that the higher the function is in the list, the more likely it is to be faulty compared to the lower ones in the list.

Finally, the type of decision tree algorithm we used is C4.5 (Witten and Frank, 2005). The C4.5 decision tree algorithm is the most widely used algorithm. It is suitable for a dataset with numerical values (e.g., see Figure 4) of independent variables, unlike ID3 decision tree algorithm (Witten and Frank, 2005) which works only with nominal values of independent variables. The C4.5 decision tree develops a model of a dataset which is then used to predict the faulty function (dependent variable) from episode rules (independent variables) in a new failure trace. The details of the C4.5 algorithm (such as calculating information gain to build a node of a tree, pruning a tree for better classification accuracy, and probability of prediction) can be found in standard text by Quinlan (1993). We show later, in Section 2.5, an example of how the C4.5 decision tree works with the MINEPI algorithm.

Several other algorithms for classification also exist, such as neural networks, support vector machines, naïve Bayes classifiers, etc. We have chosen the decision tree because in our experiments other algorithms have not yielded as effective results as the decision tree in terms of performance or accuracy. For example, we used Weka (Witten and Frank, 2005) to evaluate different algorithms on the trace dataset. Naïve Bayes, using Weka, resulted in lower accuracy than the decision tree and the neural network took quite long

for training on traces. Nonetheless, the purpose of this research is not the comparison of the classification algorithms, but to substantiate that the classification algorithms are useful in discovering faulty functions. Any classification algorithm can be used and a detailed comparison of the classification algorithms is out of the scope of this paper.

2.4 Experimental Setup

In this section, we explain the details of the setup of the controlled experiment that we conducted to evaluate F007. We have evaluated F007 on the four well-known open source UNIX utilities (i.e., Flex, Grep, Gzip and Sed) (Do et al., 2005), the Siemens suite of seven programs (Do et al., 2005; Hutchins et al., 1994) and the Space program (Do et al., 2005). All these programs were developed in the C language. The Siemens suite was developed by Hutchins et al. (1994) in the C language, and we downloaded it on March, 2008 (Siemens suite, 2008). The Space program was written for the European Space Agency in the C language and was made available by Do et al. (2005) at subject infrastructure repository (Do et al., 2005). The Siemens suite and the Space program have been used in a number of studies, e.g., Jones and Harrold (2005), Di Fatta et al. (2006), Liu et al. (2005), Bowring et al. (2004), and Wong et al. (2006). The four UNIX utilities (Do et al., 2005) are also used in different studies; e.g., Zhang et al. (2009), and are available from the subject infrastructure repository (SIR) (Do et al., 2005). This commonality of the programs under study across different studies simplifies the comparison of the F007 against other techniques[9].

In Section 2.4.1, we explain the characteristics of the dataset and in Section 2.4.2 we articulate the process of controlled experimentation on this dataset.

2.4.1 The Data Set

Table 4 shows the characteristics of the twelve different programs (including seven different programs of the Siemens suite (Hutchins et al., 1994)) used in the study. Each of

[9] We downloaded the Space program in March, 2009 (Do et al., 2005) and the UNIX utilities in August, 2010 (Do et al., 2005).

the seven programs in the Siemens suite and the Space program (Do et al., 2005) comes with an original version (deemed correct[10]), many faulty versions, and a collection of test cases. A faulty version is a variant of the original version by one fault; that is every faulty version was equivalent to one fault. A fault actually encompasses statements changed from the statements of the original program. A fault can span over multiple statements and multiple functions. In the Siemens suite, there were several instances of faults spanning across multiple functions; whereas in the Space program each fault was present in a single function. The faults in the Siemens suite were hand seeded, and in the Space program they were found during actual development.

Table 4. Characteristics of the subject programs.

Flex, Sed, Grep and Gzip are well known UNIX utilities.
In the Siemens suite: Print_tokens and Print_tokens2 are lexical analyzers; Replace is a pattern replacement program; Schedule and Schedule2 are priority schedulers; Tcas is an altitude separation program; and Tot_info is a utility for information measurement.
Space is an interpreter for an antenna array definition language written for the European Space Agency.

Releases for Flex: R1=2.4.7, R2= 2.5.1, R3=2.5.2, R4=2.5.3, R5=2.5.4.
Releases For Grep : R1=2.2, R2= 2.3, R3=2.4, R4=2.4.1.
Releases for Gzip: R1=1.1.2, R2= 1.2.2, R3=1.2.4, R4=1.3.
Releases for Sed: R1=2.0.5, R2= 3.01, R3=4.0.6, R4=4.0.7, R5=4.1.5.
Space and the seven programs in the Siemens suite have only one release.

Program	# Test Cases	LOC (excludes comments & blank lines)	# Functions	# Faulty Functions	# Faulty Versions	Total (Sum of) Failed Traces
Flex	567	8250-9831	151-169	3-12 (26)	4-16 (45)	7-362 (877)
Grep	809	8484-9041	142-150	2-4(9)	3-5(15)	11-247 (659)
Gzip	214	4032-5103	89-111	3-6(13)	3-6(16)	14-50 (99)
Sed	370	4711-9226	115-183	1-4 (10)	3-5 (18)	60-141 (465)
Space	13585	5767	136	26	34	71958
Print_tokens	4130	336	18	4	7	484
Print_tokens2	2064	343	19	4	9	2064
Replace	5542	494	21	11	32	4567
Schedule	2650	277	18	4	9	785
Schedule2	2710	249	16	6	8	275
Tcas	1608	128	9	9	40	1531
Tot_info	1052	268	7	5	23	1900

Table 4 also shows the characteristics of the four UNIX utilities. A distinct trait of the

[10] Of course, in general, there is no way to guarantee correctness. However, the program is taken as a benchmark and this then becomes the reference point.

UNIX utilities from the Siemens suite and the Space program is that they have different releases, and each release contains several faulty versions (or faults). Each fault, as in the Space program, was present in a single function. In Table 4, the second row shows release numbers for each of the releases of the UNIX utilities used in our study. We have labeled each release from R1 to R5, which will be used in the following sections. The faults in the UNIX utilities were also hand seeded (Do et al., 2005) but a specific procedure was followed to keep them realistic (Do et al., 2005). For example, the faults were inserted at the changes between source code of different releases; i.e., regression faults.

In Table 4 the first column shows the name of a program and the second column shows the number of test cases. In the UNIX utilities, the test cases were shared across releases. Third and fourth column shows the lines of code and the number of functions in a program. For the UNIX utilities, the third and the fourth columns represent minimum and maximum LOC or functions for the different releases of every program, respectively. For example, five releases of the "Flex" program have 8250 to 9831 lines of code and 151 to 169 functions. Similarly, the last three columns of Table 4 show the **minimum-maximum** number of **distinct** faulty functions, **minimum-maximum** faulty versions and **minimum-maximum** failed test cases for all the releases of every program in the UNIX utilities. A number in the bracket of the last three columns for the UNIX utilities show the **total** number of distinct faulty functions, total faulty versions and total test cases across all the releases of a program. The faulty functions in these programs are determined using the following procedure:

- If a fault has occurred across multiple functions then multiple functions are grouped together and considered as one distinct faulty function. For example, in the faulty version 11 of the program "Tcas" the functions "Own_Below_Threat", "Own_Above_Threat" and "alt_Sep_Test" have changed-statements compared to the original program. These three faulty functions were grouped together under a new joint name (including all three names) for the version 11. This is because a failure could occur because of a fault in any one of these three functions, and all

three should be identified together. Similarly there were several other instances of multiple faulty functions in the Siemens suite (Hutchins et al., 1994).

- If the same function in one (faulty) version is found faulty in other (faulty) versions (with different statements), then all these versions of a program are considered to have faults in the same faulty function. For example, function "Initialize" in the program "Tcas" is faulty across seven versions (i.e., 7,8,16,17,18,19 and 33). Similarly, this rule applies to functions across releases.

- If a fault is found in a global variable, in the source or in the header files, then the functions where that global variable is used are considered faulty. For example, the program "Tcas" in versions 13, 14, and 36 has, respectively, faults in global variables "OLEV", "MAXALTDIFF" and "DOWNWARD_RA". These variables are used in function "alt_sep_test" in the source code, and it is considered as a faulty function for versions 13, 14 and 36 for program "Tcas". Similarly, there are several similar instances of global variables in the Siemens suite.

2.4.2 The Empirical Process

We collected function-call traces (see Figure 2) for the Siemens suite using a GCC based profiling tool, called Etrace[11] (2008). For the Siemens suite and the Space program we ran all the test cases on the original version of the program as well as on all the faulty versions of the program. If the output of the same test case on the faulty and the original version differed then it is considered failed, and we collected an execution trace for that failed test case. For the four UNIX utilities, we ran the scripts (based on similar procedure) provided by the subject infrastructure repository (Do et al., 2005) to identify the failed test cases. Later, we collected the failed traces for the identified failed test cases. Following the documentation of SIR (Do et al., 2005) and other experiments (Zhang et al., 2009) for the UNIX utilities, those faulty versions (faults) were excluded

[11] Etrace has a bug which prevents it from capturing traces of the segmentation faults. We fixed it to collect such traces.

which failed on more than 20% of the test cases. This is because Do et al. (2005) concurred that faults revealed by more than 20% of the cases can be identified by testers (Do et al., 2005). So, in order to keep our results synchronized with others (Do et al., 2005; Zhang et al., 2009), we also used faults with 20% or lesser failed test cases.

In Table 4, the Faulty Versions column excludes those versions for which traces could not be captured due to non-failure of a test case or due to the exclusion condition of more than 20% of the test cases for the fault. For example, in the "Grep" program no test cases failed for all the faults (faulty versions) in release 2.4.2; that is, Table 4 excludes the fifth release for the Grep program provided by SIR (Do et al., 2005). Similarly, the "Sed" program had seven releases but no test cases failed for the release 1.18 and release 3.02. For "Gzip", no test cases failed on release 1.2.3, we have excluded it too. The versions with no failed test cases for the Siemens suite include version 4 and 9 of the program "Schedule2"; version 38 of the program "Tcas"; and version 10 of the program "Print_tokens2". In the Space program, version 1,2,32 and 34 had no failing cases. Similarly, there were several instances of faulty versions (within a release) with no failed test cases or faulty versions with more then 20% of failing cases in the Flex, Grep, Gzip and Sed programs.

We collected failed traces for these programs using Ubuntu 10.04. There could be little variations for UNIX utilities in the failing of test cases for the number of faulty versions on different platforms (Do et al., 2005). Our number of failing test cases on the faulty versions for the UNIX utilities in Table 4 mostly matches with the documentation provided by Do et al. (2005). We applied the F007 technique, using our tool (built in Java and MySQL) independently on each of the faulty programs. We used the J48 algorithm API in the Weka library (Witten and Frank, 2005) for the implementation of the C4.5 decision tree algorithm

Recall from Section 2.3.1 that in order to generate an optimal number of episodes, higher length episodes were generated by setting the minimum frequency to 1, the minimum confidence to greater than 0, while varying the window width and episode length to different values. Also recall that in MINEPI (Liu et al., 2005) these parameters (window

width, episode length, frequency, confidence) are selected by a user. Therefore, to find the best accuracy of predicting the faulty functions, we varied window widths to length 3, 5 and 7; and episodes to length 1, 2 and 3. We stopped building episodes with higher window widths or higher lengths when accuracy stopped improving (discussed in Section 2.6). In summary, we generated: (i) the episodes of length 1; (ii) the serial and parallel episodes of length 2 and 3; and (iii) the rules (see Section 2.3.1) for serial and parallel episodes of length 2 and 3 for every trace of a program and for every window width. Finally, the C4.5 decision tree algorithm (using one-against-all approach (Witten and Frank, 2005)) has been trained on episode rules of every length for each window width. This was done to discover the best combination of episode rules and the window width for a program in discovering the faulty functions.

To train the C4.5 algorithm, the dataset was divided into four different stratified parts (Witten and Frank, 2005). In the stratification of data, each of the categories of dependent variables (different faulty functions in our case) is represented in approximately the same ratio in each new part as it is in the original dataset. We used three parts (approximately 75% of dataset) for training the C4.5 decision tree algorithm (using one-against-all approach) for each length of different episode rules (serial/parallel) in a window width, and used one part (approximately 25%) for testing. We repeated the above procedure for training and testing three times by using a different part for testing and three different parts for training each time. The accuracy of the prediction was then averaged for each of the three repetitions. This is called three fold cross validation (Witten and Frank, 2005). This was done to avoid any bias in the accuracy of prediction if only specific part is used for training and testing. Literature (Witten and Frank, 2005) recommends selecting more than 50% of data for training to avoid under training a classifier and use 3 to 10 fold cross validation – if the dataset is not very large.

2.5 Executing F007

In this section, we describe an illustrative example of identifying faulty functions in the Siemens program “Tot_info” by using F007. In Figure 5 (part a), rows represent failed traces of 23 faulty versions of “Tot_info” (to save space only a random selection of example episode rules and failed traces are shown). The column headers show episode

rules of length 2 (generated using window width of 3) and actual faulty functions for the failed traces of a particular version. There were 37 episode rules, 1900 fail traces and *five* distinct faulty functions. Each trace is represented by a trace number (e.g., T1000) with its corresponding version number prefixed (e.g., V10_T1000). A cell in Figure 5 shows the confidence of an episode rule in a trace, and the cell of the last column shows the name of faulty functions for that trace.

	Episode Rules of Length 2							
	gcf => (gcf < InfoTbl)	gser => (gser < InfoTbl)	InfoTbl => (InfoTbl < LGamma)	LGamma => (LGamma < QChiSq)	.....	QChiSq => (QChiSq < gcf)	QChiSq => (QChiSq < QGamma)	Faulty Functions
V10_T1000	0.5	0.2	0.4	0.45	.	0.28	1	InfoTbl
V11_T100	1	0	0.5	0	.	0.5	1	gser
V12_T101	0	0.5	0.6	0.5	.	0	1	LGamma
V11_T1005	0	0.33	0	0	.	0	1	gser
.....								.....
V23_T1002	1	0.5	0.66	0.5	.	0.33	1	gcf

(a) Original dataset for all categories

V10_T1000	0.5	0.2	0.4	0.45	.	0.28	1	others
V11_T100	1	0	0.5	0	.	0.5	1	gser
V12_T101	0	0.5	0.6	0.5	.	0	1	others
V11_T1005	0	0.33	0	0	.	0	1	gser

(b) Dataset for category "gser" against all others.

Figure 5: Length 2 episode rules, faulty functions and traces of 23 versions of "Tot_info" from the Siemens suite.

Recall (from Section 2.3.2) that in the one-against-all approach (Witten and Frank, 2005), a dataset with M faulty functions (category of dependent variable) is decomposed into M (total categories) new datasets. Therefore, we transformed the dataset of Figure 5 (part a)

into five new datasets such that each new dataset contained traces labeled only for one faulty function. The rest of the traces for other faulty functions were labeled as “others”. An example of a faulty function “gser” against all “other” faulty functions is shown in part b of Figure 5. It again shows a random selection of example traces (to fit space). The columns for part ‘b’ of Figure 5 are the same as for part ‘a’ of Figure 5.

In order to evaluate the prediction accuracy, the original dataset was first divided into two parts: training (75%) and testing (25%). This 75% of the original training data was actually decomposed into five new datasets (using one-against-all approach) as shown in part b of Figure 5.

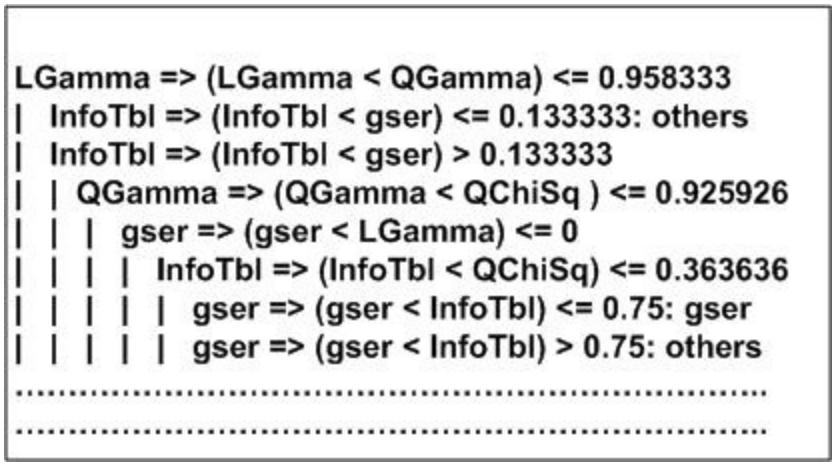


Figure 6: The C4.5 decision tree model for the faulty function “gser” of “Tot_info”.

A separate decision tree was trained on each of the five new datasets. An excerpt of a trained decision tree generated for part b of Figure 5 is shown in Figure 6. This tree was obtained by applying the J48 algorithm in the data mining tool Weka (Witten and Frank, 2005) which was the implementation of the C4.5 decision tree algorithm. We will not show the steps of generation of the C4.5 algorithm (Witten and Frank, 2005) due to complexity, size and cluttering of text - however the reader may refer to the standard texts (Quinlan, 1993; Witten and Frank, 2005) for details.

A row in Figure 6 contains an episode rule, its confidence, and a name of faulty function after a colon sign if any. An episode rule with the confidence value represents the node of

a tree and faulty function names after the colon sign represents the leaf of a tree. The discovery of a faulty function was done by traversing this trained tree (like If-Then-Else statements) according to the confidence values of a trace in a test set. For example the decision tree of Figure 6 shows that if in a failed trace, the confidence value of an episode rule "LGamma => (LGamma < QGamma)" is less than or equal to "0.958333" and the confidence value of "InfoTbl => (InfoTbl < gser)" is less than or equal to "0.13333", then the faulty function is "others" (that is not "gser").

Similarly, a total of five different decision trees according to the one-against-all approach were trained. Every failure trace in a test set was input to the five decision trees, and each of the trees predicted a faulty function for the trace with a probability. The probability of prediction in the C4.5 algorithm is determined by measuring the number of training instances correctly classified at a leaf and dividing it by the total number of instances correctly and incorrectly classified at that leaf (Quinlan, 1993). Functions were then arranged in a list in the decreasing order of the probability. An example of a ranked list of faulty functions for a trace "T1013" of version 1 of "Tot_info" is shown in Figure 7. In this case, "InfoTbl" was the faulty function and was ranked at position 1, as it had the highest probability produced by one of the five C4.5 decision trees. The fifth decision tree predicted "others" as faulty function, which can be ignored because it means other functions. F007 was, similarly, applied to other episode rules and window widths of all the subject programs in Table 4.

	Tot_info V1_T1013	
	Function	Probability
Rank 1	InfoTbl	0.708
Rank 2	gser	0.27
Rank 3	gcf	0.08
Rank 4	LGamma	0.02

Figure 7: Faulty function ranking for trace "T1013" of version 1 of program "Tot_info".

Similarly, we identified faulty functions in every failed trace of the test set and measured the accuracy of the identification of faulty functions for the test set. Finally, we repeated

the above process two more times (three in all) every time with a different 25% test set and 75% train set (according to the three fold cross validation). The accuracy on the test set was then averaged. As mentioned in Section 2.4.2, this is called three fold cross validation and the results are shown in the next section.

2.6 Results

In this section, we show the results of evaluating F007 on the programs in Table 4. In Section 2.6.1, we identify the best episode rules (patterns) of function-calls (using the decision tree) in identifying faulty functions. This section allows us to determine which episode rule should be used to determine the faulty functions. In Section 2.6.2, using the best episode rule obtained in Section 2.6.1, we evaluate F007 using a realistic scenario: (a) identify the faulty functions in successive releases using the traces of previous releases, and (b) use a minimal number of traces associated with one fault of a function to identify the same faulty functions with another fault or same fault. This section helps us in determining that different faults in the same function can be diagnosed correctly, and the same faults with the knowledge of few traces can also be identified correctly. In Section 2.6.3, we explain that rules generated from the decision tree can be useful in diagnosing fault proneness of a function from the perspective of related functions. In Section 2.6.4, we show that only "entry" or "exits" are sufficient to discover faulty functions from the function-call level trace.

2.6.1 Episode Rules

In this section, we first determine in Section 2.6.1.1 the difference between accuracies of different episode rules using statistical tests and identify the best episode or episode rule to determine the faulty functions. Later in Section 2.6.1.2 we show the execution statistics of F007 using the best episode or episode rule.

2.6.1.1 Evaluating Episode Rules

In this section, we actually evaluate the effect of episode rules (patterns) of function-calls (using the decision tree) in identifying faulty functions in failed traces. We use three fold cross validation in estimating the accuracy of identifying faulty functions. Table 5 shows

the results of applying F007 to the programs shown in Table 4. We have randomly selected a handful of programs from Table 4 to be shown in Table 5 to avoid cluttering the text. In Table 5 columns depict program name, the type of episode rule, episode length, and percentage of failed traces successfully diagnosed on reviewing the first function in the list, using the window width 3, 5 and 7. For example, row three shows that the faults in 73.553%, 72.933% and 70.041% of the failed traces in the test set were successfully identified after reviewing the first function in the list obtained using serial episode rules of length 3 of the program "Print_tokens" and window widths 3, 5 and 7 respectively. Row three also shows that these are average accuracy values, on three different test sets, obtained using three fold cross validation.

Table 5: Faulty functions prediction accuracy (in percentage) for failed traces of the programs using window widths 3, 5 and 7.

Programs	Episode Rule Type	Length	Win(w)= 3	Win(w)= 5	Win(w)= 7
Print_ tokens	NA	1	74.380	74.380	74.380
	Serial/Parallel	2	73.347	68.801	70.041
	Serial	3	73.553	72.933	70.041
	Parallel	3	73.347	71.900	68.595
Print_ tokens2	NA	1	61.773	61.773	61.773
	Serial/Parallel	2	56.298	57.364	55.523
	Serial	3	56.346	57.655	58.624
	Parallel	3	57.9	57.509	57.46
Replace	NA	1	65.447	65.447	65.447
	Serial/Parallel	2	65.963	65.611	64.932
	Serial	3	64.861	65.096	65.518
	Parallel	3	63.97	63.806	64.838
Grep (R3)	NA	1	95.546	95.546	95.546
	Serial/Parallel	2	99.595	98.380	98.380
	Serial	3	99.190	97.571	97.976
	Parallel	3	98.785	97.976	98.380
Sed (R3)	NA	1	89.361	89.361	89.361
	Serial/Parallel	2	95.745	93.617	93.617
	Serial	3	95.035	94.326	94.326
	Parallel	3	94.326	95.035	92.198
Gzip (R1)	NA	1	90.0	90.0	90.0
	Serial/Parallel	2	90.0	90.0	90.0
	Serial	3	92.0	92.0	90.0
	Parallel	3	92.0	90.0	90.0

In Table 5, the prediction accuracy for the length 2 parallel episode rules and the serial episode rules are shown together because they yield the same episode rules and the same

accuracy. Also, in Table 5, episodes of length 1 have no types and they are also independent of window widths (i.e., win(w) = 0), but we have shown them together with other window widths. In terms of the UNIX utilities, we have selected one of the releases with the largest number of failed traces. This is because we are trying to evaluate the accuracy of episode rules, and the results of one release will hold for others too. In the next section we shall show the results on all the releases.

Recall from Section 2.3.1 that larger window widths result into more episode rules and consume more space and time. However, it can be seen in Table 5 that the difference in accuracy between window widths is marginal. We have omitted the information on time and the number of episode rules to avoid cluttering of text in Table 5. In order to determine if there is any significant difference in the accuracy between different window widths, we conducted the Wilcoxon signed-rank test (Marques de Sá, 2003). We selected the Wilcoxon signed-rank test because the data, part of it shown in Table 5, did not follow the normal distribution. We analyzed the normality of data using Shapiro-Wilk test (Marques de Sá, 2003) by setting the alpha level (or level of significance) to 0.05. The Shapiro-Wilk test for 48 data points of 12 programs at window width 3 resulted into W=0.8721 and p=0.00004 (< 0.05). This means that the data distribution is not normal because the null hypothesis that data are drawn from normal distribution is rejected as $p < 0.05$. The histogram of the data points also showed a positively skewed distribution, confirming that the data is not normal. Similar results of non-normal distribution were also obtained for data points of win(w) = 5 and win(w) = 7.

We first conducted the Wilcoxon signed-rank test between the window width 5 and the window width 7 with the null hypothesis: “there was no significant difference between classification accuracy of win(w) = 5 and win(w) = 7”. We again set the alpha level to 0.05 as at this level we can reduce the risk of type 1 error (false positive). The Wilcoxon signed rank test for 48 (i.e., for 12 programs) matched observations did not result in significant difference between “win(w) = 5” (M=73.477, SD=13.018) and “win(w) = 7” (M=73.346, SD=12.935) with z=0.227 and (two-tailed) p=0.820. This provides the evidence ($p > 0.05$) that the null hypothesis could not be rejected and the fault prediction accuracies of “win(w) = 5” and “win(w) = 7” were identical.

Similarly, the Wilcoxon signed-rank test between: (a) "win(w) = 3" (M=73.796, SD=13.299) and "win(w) = 5" (M=73.477, SD=13.018) produced z = 1.708 (two tailed) p=0.088; and (b) "win(w) = 3" (M=73.796, SD=13.299) and "win(w) = 7" (M=73.346, SD=12.935) generated z=2.047 (two-tailed) p=0.041. In the case of win(w) = 3 and win(w) = 5 (case 'a'), the accuracies with different window widths were identical (i.e., $p > 0.05$). In the case of win(w) = 3 and win(w) = 7, the accuracies were not identical (i.e., $p < 0.05$); in fact, the accuracy to discover the faulty functions at win(w) = 7 had actually started decreasing. Thus, the use of window width win(w) = 3, which was also cheaper compared to higher window widths, was satisfactory for classifying the failed traces.

We can also observe from Table 5 (win(w) = 3) that accuracy also varies among the episode lengths. To determine if there is any improvement in the accuracy between different episodes rules, we conducted the Wilcoxon signed rank test between episodes of length 1 and the higher episode rules within win(w) = 3. The Wilcoxon signed rank test with 12 observations between: (a) the episodes of length 1 (M=74.285, SD=12.325) and the serial/parallel episode rules (M=73.940, SD=13.927) of length 2 resulted in z=0.178 and (two tailed) p=0.859; (b) the episodes of length 1(M=74.285, SD=12.325) and the serial episode rules of length 3 (M=73.667, SD=14.364) resulted in z=0.628 and (two tailed) p=0.530; and (c) the episodes of length 1 (M=74.285, SD=12.325) and the parallel episode rules of length 3 (M=73.293, SD=14.245) yielded z=0.549, (two tailed) p=0.583.

In all these cases the value of p is significantly higher than significance level of 0.05, substantiating that there is no significant difference between the accuracy of the episodes of length 1 and the higher length (serial or parallel) episode rules. This implies that the episodes of length 1, which are just single function-calls, are not only cost-effective to generate, but also yield equivalent (or better) fault prediction accuracy than higher length episode rules. It should be noted that our experiments are exhaustive and include several combinations of episode (function patterns) types, window widths and episode lengths, but these experiments suggests that the same results can be obtained using length 1 episodes (single function-calls) if the decision tree is used with them for predicting faulty functions in failed traces. We also experimented with the length 1 episodes using the frequency values instead of the confidence values. The results were similar, and no

difference existed. We have not shown the data points here because it does not have any further impact on the results of the episodes.

2.6.1.2 Execution Statistics

In Table 6, we show the best faulty function prediction accuracy of F007 on each of the twelve programs, obtained using episode of length 1. Table 6 also shows the number of episodes, average size of a failed trace, time taken per trace to generate episodes of length 1 (including I/O), and the accuracy of finding faulty functions in the failed traces on reviewing the first function in the list (generated using F007). An interesting characteristic that can be observed from Table 4 is that each program has a different number of functions, but in Table 6 the accuracy of identifying faulty functions remains similar. The average accuracy for the twelve programs is appriximately 70% on reviewing only the first function in the list. This shows that the faults in the same function occur with similar sequences of function-calls. Thus, if 20% of the code (i.e., functions, components) is responsible for 80% of the faults then majority of the faults can be identified by failed traces of previous faults in the same function.

Table 6: Execution statistics of the best episodes.

Program	# of Episodes	Avg. size of a trace (KBs)	Time per trace with I/O (sec)	Accuracy
Print-tokens	19	43.795	0.364	74.3
Print-tokens2	20	35.622	0.277	61.773
Replace	22	22.227	0.268	65.447
Schedule	19	16.521	0.2194	71.488
Schedule2	17	33.028	0.268	60.363
Tcas	10	1.016	0.110	73.481
Tot_info	8	1.509	0.014	68.482
Space	125	33.62	0.235	73.6
Flex	143	516.0	1.809	60.905
Grep	90	107.35	0.431	89.919
Gzip	58	1294.9	4.047	66.666
Sed	62	43.98	0.369	85.611

The execution time in Table 6 was obtained by performing experiments on a 3GHz CPU, with 3 GB of RAM. This time measurement involved the use of a network drive to read traces, running of Java application (i.e., F007's implementation) in NetBeans, and the use

of MySQL database to store episodes on a local hard disk. This time could vary depending on the proper database configurations and implementation techniques used for programming. We optimized execution time by using bulk inserts and bulk reads when accessing disk and database system, but we believe that this time can be improved further.

Nonetheless, processing single function-calls (length 1 episodes) and their frequencies is a trivial task – unlike patterns of function-calls (higher length episode rules). It should be noted that this length 1 episode generation (time) was only required to be done once for historical traces. It can also be observed that length 1 episode generation time increased linearly with the trace size. In addition to this time, there was a time required to generate the C4.5 decision tree model, which was dependent on the Weka (Witten and Frank, 2005) API implementation. The maximum time for the C4.5 tree generation was approximately 5 minutes for the Space program. In practice, the decision tree model must also be generated once, but should be updated when traces with new faulty functions are included in the database.

2.6.2 Identifying Faulty Functions in Failed Traces using Minimal-earlier Failed Traces

In this section we first show that by using a small percentage of the failed traces for training, F007 can identify the faulty functions in the rest of the failed traces (Section 2.6.2.1). Secondly, we show that F007 can identify faulty functions in the following releases using the traces of previous releases (Section 2.6.2.2). Thirdly, we deomonstrate that how much would be the effort in statements when identifying faulty functions using F007 (Section 2.6.2.3).

2.6.2.1 Using a smaller percentage of traces for training and a larger for testing

In Section 2.6.1, we used three-fold cross validation to identify the best episode in predicting faulty functions. However, realistically speaking, it is not feasible to use 75% of the failed traces for training to identify faulty functions in the rest of the 25% failed traces. In reality only a small percentage of failed traces will be available initially to

identify faulty functions in the rest of the failed traces. For example, once a software application is deployed, initially there will be few traces from the deployed instances of the software application. A realistic case would be to use few initial failed traces to predict faulty functions in the rest of the upcoming traces from the field; later on, new failed traces could be used for training as they get resolved. Therefore, we now evaluate F007 by using 25% or less failed traces for training to identify the faulty functions in the rest of the failed traces. If 50-90% of the field failures are rediscoveries of previous faults and 20% of the faulty functions (code) is causing 80% of the faults with similar function-call sequences (see Section 2.6.1), then F007 (with 25% test set) should be able to identify faulty functions in the majority of the failed traces.

We evaluated our approach following the similar graphical convention used by Jones and Harrold (2005), Wong et al. (2007) and Di Fatta et al. (2006). This makes the comparison simpler with others and results easier to interpret. Hence, we computed a score for each failed trace as the percentage of program (i.e., functions or statements) needs to be reviewed to find the fault. Horizontal axis (X-axis) represents the percentage of a program that needs to be examined and X-axis is divided into segments. Each segment is ten percentage points except for the first segment which is divided into 1 percentage points; i.e., 1-10% segments are divided into 1 percentage points and 90-100% segments are divided into 10 percentage points each. The vertical axis (Y-axis) measures the cumulative percentage of failed traces that achieve a score within a segment.

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

Equation 1: Estimating program review effort in functions.

For example, in Figure 8, the Y-axis shows the cumulative percentage of failed traces, and the X-axis represents the percentage of the program to be examined in identifying faulty functions. The X-axis is measured by the percentage of functions reviewed, in the chronological order from the list generated by F007 (see Section 2.5), in identifying faulty functions in the failed trace of a program. This is shown in Equation 1, which

shows percentage (%) of program (in terms of functions) to review for a failed trace is the functions reviewed, divided by the total number of functions.

In Figure 8, the point (1, 77) (first point) on the (red or marked by squares "■") series called "Flex" shows that the faulty functions in 77% of the failed traces of the five releases of the Flex program were discovered by reviewing 1% (≈ 1 function) or less of the code. Similarly, in the same series, the faulty functions in 95% of the failed traces were discovered by reviewing 2% (≈ 2 functions) or less of the code. The results, in Figure 8, for the Flex program are obtained by training F007 on 25% of the failed traces of a release and predicting faulty functions in rest of the 75% failed traces of that release. This process is repeated for each release of the Flex program and the score (using Equation 1) of each failed trace (in the test set) of each release is also calculated. The percentage of the failed traces for the Flex program is measured by summing the number of failed traces of all the releases of Flex that fall within each segment (on X-axis) and dividing them by the total number of failed traces. Finally, the results are then shown as the cumulative percentage of failed traces on Y-axis. Similarly, Figure 8 also shows the accuracy of prediction of faulty functions for all the releases (see Table 4) of Grep, Gzip and Sed programs. The results for the Grep, Gzip and Sed are obtained in exactly the same manner to the results of the Flex program.

Figure 8 also shows the accuracy of identification of the faulty functions in the failed traces of the Siemens suite (shown by pink series or marked by "—"). In the Siemens suite, there were seven programs, each with one release. The results in Figure 8 are obtained by training F007 on the 25% failed traces of each program and using the rest of the 75% failed traces as test set for the respective program. The cumulative percentage of failed traces is measured in a similar manner to the Flex program; except the Siemens suite has seven programs whereas the Flex program has five releases. Finally, Figure 8 depicts the result on the Space program too. The Space program has only one release but a very large number of failed traces (see Table 4). We therefore trained F007 only on the 10% of failed traces of the Space program and tested on the remaining 90% of the failed traces (Refaat, 2007). This is because if a data set is very large, a minimum of a 10% sample of large data is considered as a good estimate of original data including training

and testing (Refaat, 2007), but to show we can identify the faulty functions using minimal traces we used only 10% of the traces for training.

An important thing to note here is that we used 10% of the traces (for training) for the Space program and 25% traces for the UNIX utilities and the Siemens suite. In the case of the Space program, there were about 72,000 failed traces and the use of 10% traces still resulted into approximately 7000 traces for training. On the other hand, all the UNIX utilities had fewer than 500 failed traces and the most of the programs of the Siemens suite had fewer than 2000 failed traces. Due to fewer failed traces of the UNIX utilities and the Siemens suite, we selected 25% of their traces for training F007. The reason lies in the fact that the decision tree requires a reasonable number of traces for training; for example, literature (Witten and Frank, 2005) recommends selecting more than 50% of data for training when data set is small—we still use less than 50%.

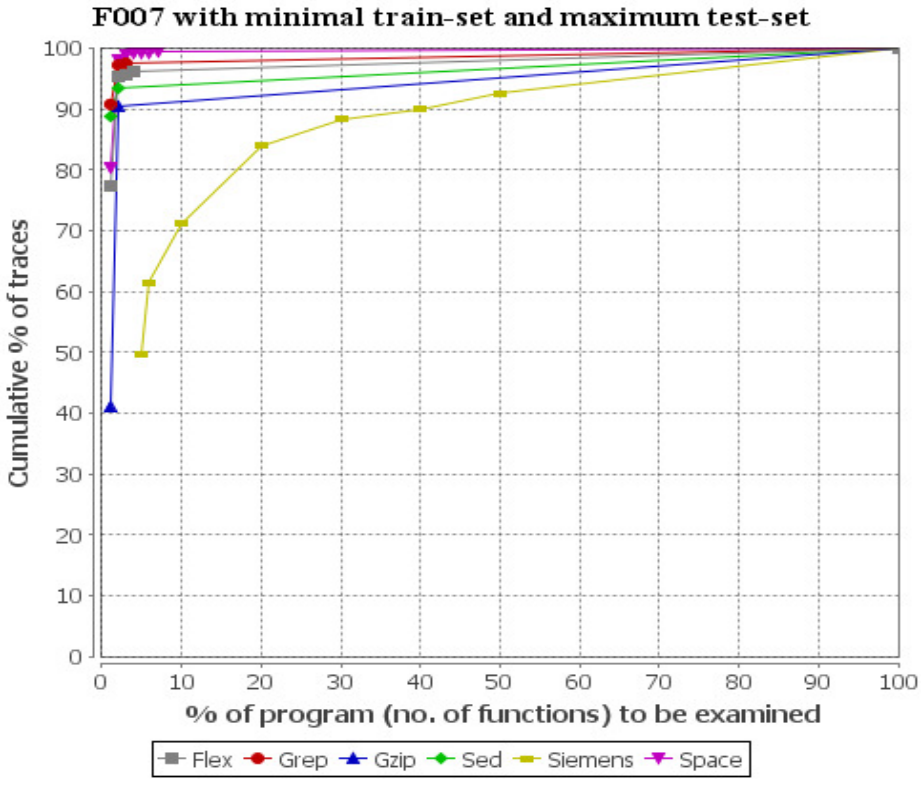


Figure 8: Accuracy of F007 on: all the releases of Flex, Grep, Gzip and Sed programs; the seven programs of the Siemens suite; and the Space program.

It should also be noted that the faulty functions in the majority of the failed traces can be identified by reviewing top one to three functions. For example, in terms of the Siemens suite, faulty functions in approximately 72% of the failed traces can be identified by reviewing 10% of the program (≈ 1.5 function) and about 85% of the failed traces can be diagnosed by reviewing approximately 20% of the program (≈ 3 functions). In all other programs faulty functions in approximately 90% of the failed traces can be identified by reviewing top 1-3 functions (i.e., approximately 3% or less of the program). Also note that, straight lines at the end of a series till the 100% traces when there are no more points visibile on a series mean that: F007 does not result in any more predictions of faulty functions in traces and a developer identifies faulty functions by random guesses till the 100% traces. For example, in the case of the series "Siemens" in part 'a' of Figure 8, 92% of the failed traces were resolved correctly by reviewing 50% of the program using F007 after which a developer randomly gusesses the faulty functions in the remaining 8% of traces.

In Section 2.1, we mentioned our initial observations in the Space program (see Figure 2), when the same or different fault occurs in the same function then the function calls exhibit similar patterns. Our results in Section 2.6.1 and in Figure 8 also confirm that the same faulty functions do have similar sequences of function-calls, because we have used a smaller percentage of traces for training and a larger for testing.

2.6.2.2 Identifying faulty functions across releases

To strengthen the finding, different faults in the same function indeed occur with similar sequences of function-calls, we trained F007 on the failed traces of earlier releases of the UNIX utilities to identify faulty functions in the following releases -- the faults in one release are different from the faults in other releases.

The results are shown in Figure 9, which are obtained in a similar manner to Figure 8; that is: (a) the training set contains the failed traces of past releases of the program; (b) the test set includes the traces of the following release of the program; (c) the score of each failed trace is again measured using Equation 1; and (d) the percentage of failed traces for a segment on X-axis is the percentage of failed traces of the (same level release

in) test-set of all the programs that fall within each segment. For example, the series "using release 1-3 for release 4" shows that by training F007 on the failed traces of release 1, faulty functions in approximately 60% of the failed traces can be identified in release 4 (test-set). This identification requires the review of approximately 3% or less of the code, and it is the accuracy of identifying faulty functions in the traces of release 4 using the traces of release 1 to release 3 for all the four programs in the UNIX utilities (i.e., Flex, Grep, Gzip and Sed). Similarly, Figure 9 shows the result on other releases of the UNIX utilities by using the traces of all the preceding releases as the training set and the following releases as the test set.

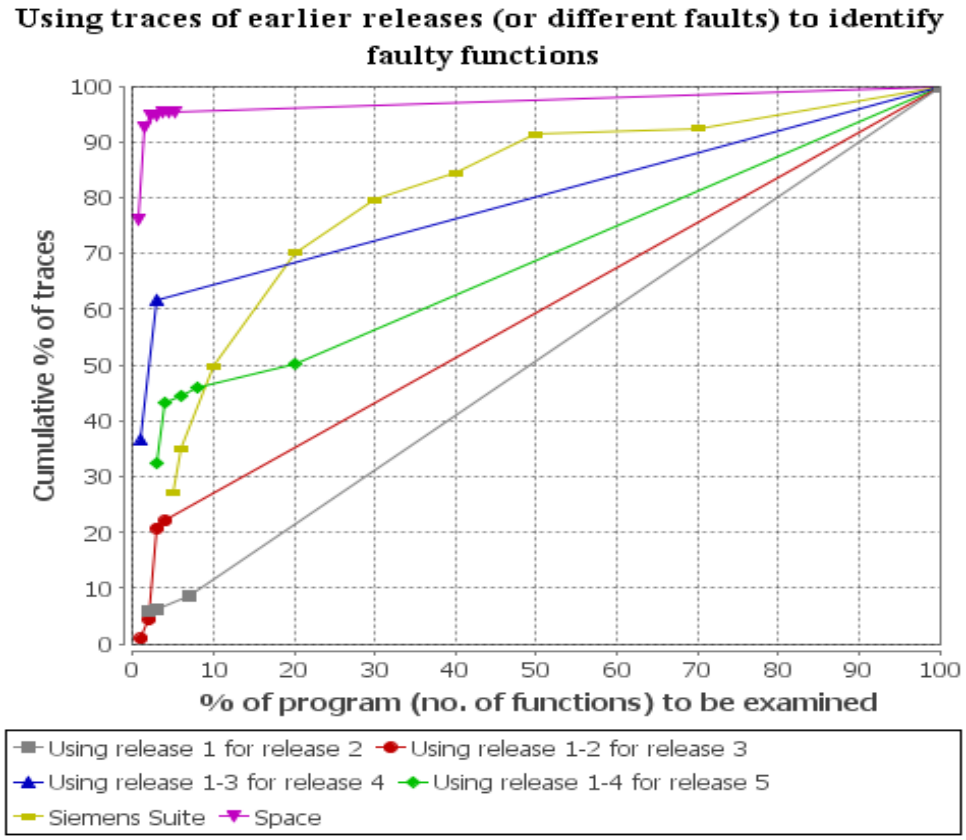


Figure 9: Using traces of earlier releases or different faults for training F007 and testing F007 on successive releases.

Figure 9 also shows the accuracy of identifying faulty functions in the Siemens suite and the Space program—each of which has only one release. In terms of the Siemens suite, we used only 25% of the training set by removing the trace records of more than one fault

(faulty versions in the Siemens suite) for a function (if existed). In other words, the training dataset contained trace records for only one fault (version) for each function. Traces of the same or other faults in the same function were kept in the test set. The results for the Siemens suite in Figure 9 show the accuracy of identifying the faulty functions for the seven programs in the same manner to Figure 8. Similarly, we applied this procedure to the Space program. The Space program contained approximately 70,000 failed traces of overall faults and taking advantage of the large number of the failed traces we used only 1% of the traces for training F007 and the rest of the 99% for testing. Also, like the Siemens suite, we removed the traces of more than one fault in the same function. The results for the Space program are also shown in Figure 9.

It can be observed in Figure 9 that the accuracy of identifying the faulty functions in release 2 using release 1 of the UNIX utilities is very low compared to the identification of the faulty function in other releases of the UNIX utilities. This is because the majority of the functions found faulty in the release 2 were not found faulty in the release 1. So, those functions could not be predicted, whereas the same faulty functions (with different faults) were predicted correctly –i.e., less than 10% of the code (function) review. Similarly, in the case of identification of the faulty functions in the traces of release 3 using the failed traces of releases 1-2, about 23% of the failed traces were correctly diagnosed on the review of 5% of the program. In the rest of the cases, faulty functions were not found in the earlier releases. In the case of release 5, the accuracy is about 47% on the review of 8% of the code, and in the case of release 4 of the UNIX utilities, the accuracy of identification of faulty functions is the highest; i.e., 60% on the review of 3% of the program.

Note that, in the case of the UNIX utilities, the accuracy improved as the traces of more and more releases were used for training F007 to identify the faulty functions in the subsequent release. For example, from release 2 to release 4 it increased by 50 percent and in release 5 it improved by 40% from release 2. This is because the training-sets had more faulty functions which were common to the subsequent release. Also, note that mostly the identification of the faulty functions in the subsequent releases was made on reviewing 10% of the program or less; i.e., first few functions in the list. Similarly, in the

case of the Siemens suite and the Space program, faulty functions in the majority of the failed traces were identified by reviewing first two to three functions. In the Siemens suite, 20% of the code is approximately equivalent to three functions, because the Siemens suite is a collection of small programs. Furthermore, this identification in Figure 9 is done by training F007 on either traces of different faults from previous releases (in the cases of the UNIX utilities) or traces of at least one fault in the same function (in the case of the Siemens suite and the Space program). Thus, this implies that if (different) faults are in the same function then they can be identified accurately, provided the traces of at least one fault in the same function are present in the training set.

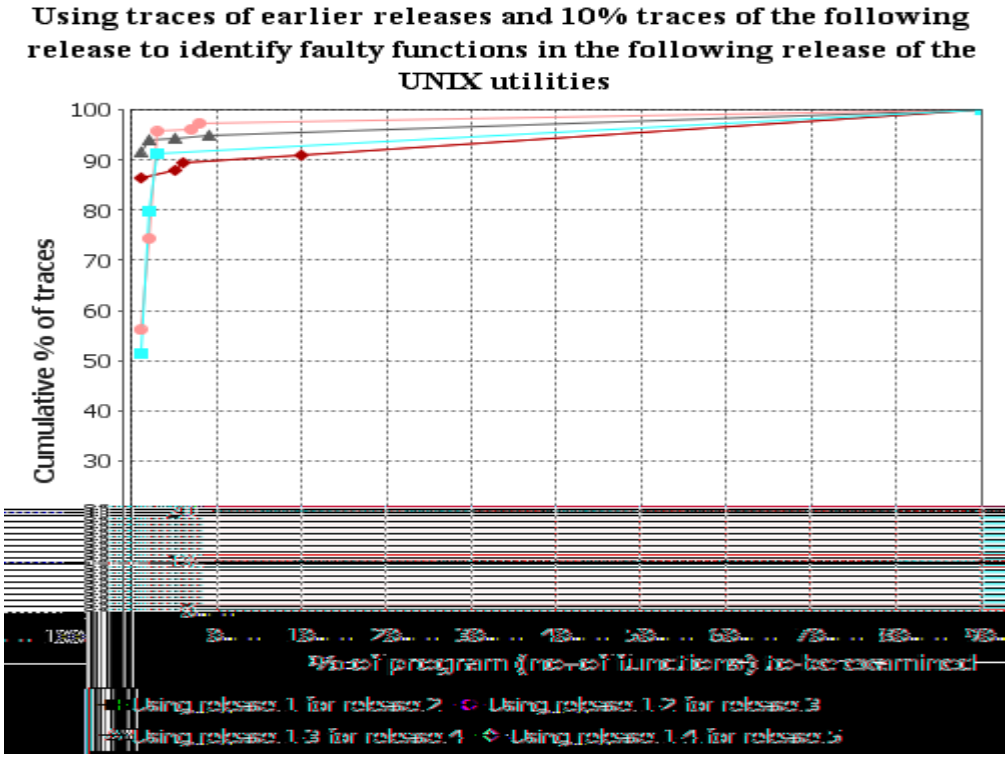


Figure 10: Using traces of earlier releases and 10% traces of the following releases to train F007 and identify faulty functions in the rest of the traces of the following release.

In some traces, both in Figure 8 and Figure 9, we were not able to identify faulty functions at all and the whole program had to be reviewed. However, in the majority of traces, if faulty functions were present in the training set, we were able to identify faulty functions correctly in both of Figure 8 and Figure 9. This implies that different faults in

the same function do occur with similar sequences of function-call, but up to a certain limit. This is because in some cases the patterns of function-calls for faults in one function matched with the patterns of function calls for faults in another function. They were not always differentiable but mostly we were able to distinguish faulty functions by reviewing only the first few functions.

In Figure 10, in a similar manner to Figure 9, we show the accuracy of identifying faulty functions across releases in the UNIX utilities. This time we also used 10% of the failed traces of the succeeding releases along with the failed traces of the preceding release to train F007. It can be observed in Figure 10 that the faulty functions in approximately 70-90% of the failed traces were identified by reviewing only 5% or less of the code (functions). If 50-90% of the field failures are rediscoveries of the same fault (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) then this can be highly significant in identifying the faulty functions in the field traces. We also know that 20% of the code is responsible for 80% of the faults (Gittens et al., 2005; Ostrand et al., 2005). Therefore, if the faults in the same function occur with similar sequences of function-calls (as we have verified from the results in Figure 9) and 50 to 90% are rediscoveries then we can identify faulty functions in 80-90% of the field failed traces using F007 (provided the training-set contains the traces of faulty functions). This is what we observed in Figure 10, that is, approximately 70-90% of the failed traces were identified on reviewing first few functions by using F007. In the case of Figure 9, a faulty function was identified with high accuracy (i.e., by reviewing 10% of the code) if it was also faulty in earlier releases, and the accuracy improved with the number of releases—implying that 20% of the code is causing majority of the faults. Finally, after identifying the faulty function, previously known faulty statements for that faulty function can be used for inspection.

2.6.2.3 Measuring statements-effort

So far, we have shown the accuracy of identifying faulty functions by using the percentage of functions reviewed as the code reviewed. However, the sizes of functions vary in a program, and functions with large sizes could account for the majority of the code and hence large numbers of faults. Therefore, in order to quantify the effort of a

developer in terms of number of statements, we summed all the statements of a function that would be reviewed by a developer up to the faulty function in the list of suspected functions generated by F007. For example, if the fourth function in F007's list is the actual faulty function then we summed the number of statements of all the four functions in estimating the effort. This is shown in Equation 2.

$$\begin{bmatrix} \%\ of\ \ program \\ to\ \ review \end{bmatrix} = \left[\frac{\sum Statements\ of\ a\ Function\ \ \mathrm{Re}viewed}{Total\ Satements} * 100 \right]`$$

Equation 2: Estimation effort in statements.

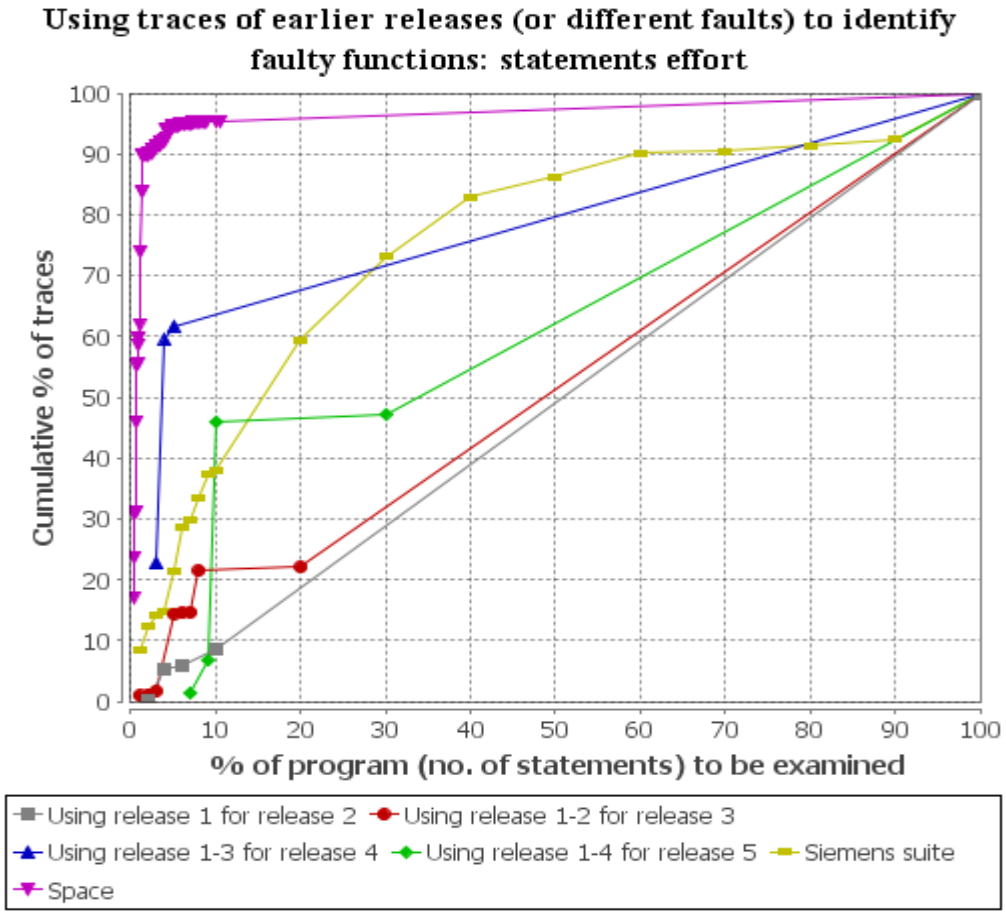


Figure 11: Statements-effort using F007 in identifying faulty functions.

In Figure 11, we show effort of a developer in identifying faulty statements using F007 across releases of the UNIX utilities, on the Siemens suite, and on the Space program.

Figure 11 is similar to Figure 9 except the effort to identify a function is estimated in statements using Equation 2. Similarly, Figure 12 shows the same results as in Figure 10, but the effort is estimated in terms of statements. Figure 11 and Figure 12 show that there is not much difference between the effort in statements and that in functions in Figure 9 and Figure 10. In fact, the effort in statements seems to be proportional to the effort in functions. This implies that in commercial or professional programs the sizes of the functions are not distributed in distinctly large to small proportions, but that they are distributed mostly in closely related sizes. However, Figure 11 and Figure 12 only represent statements of faulty functions. In order to validate this we drew functions vs. size graph of randomly selected releases of the four UNIX utilities. This is shown in Figure 13, where we can observe that apart from few outliers mostly the sizes of functions remain close to each other.

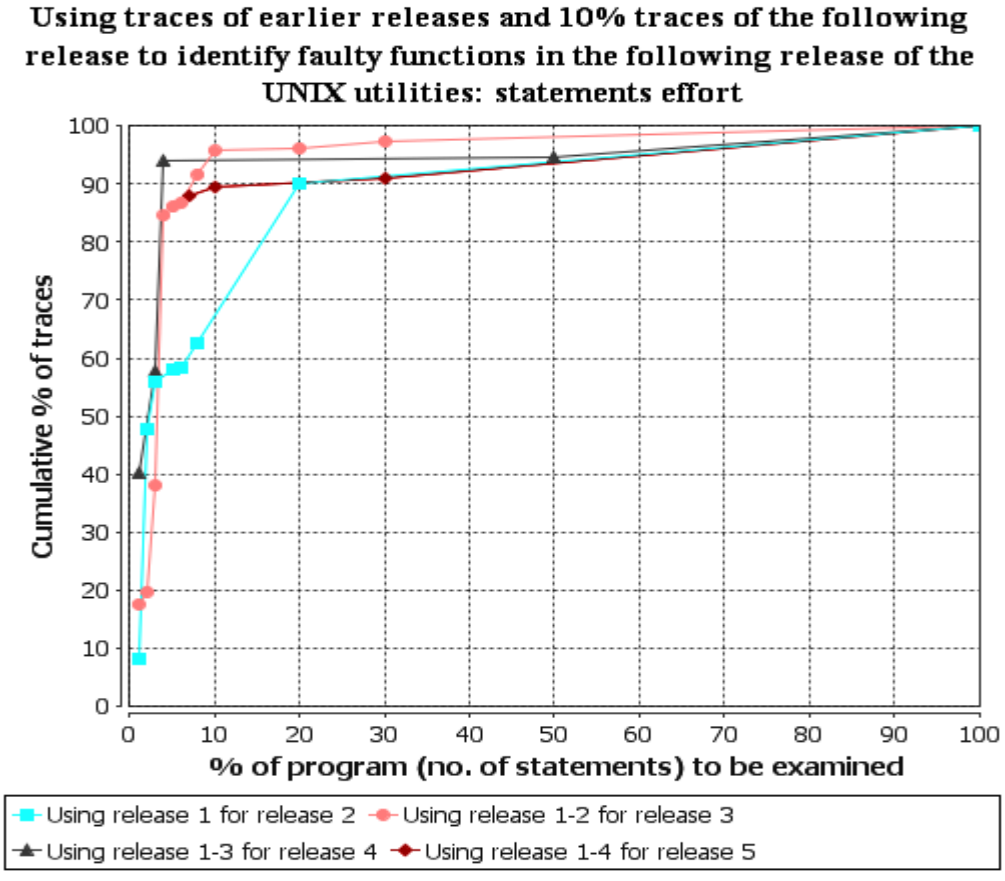


Figure 12: Statements-effort in using traces of earlier releases and 10% traces of the following releases to train F007 and identify faulty functions in the rest of the traces of the following release.

In Figure 11 and Figure 12, the estimation of effort in terms of statements is actually going to be lower than what we have shown. This is because we have summed all the statements of a function, and the programmer using the context of the fault can skip a function or may jump to another function after reviewing a few statements. Thus, in reality the effort in statements would be better than the shown here.

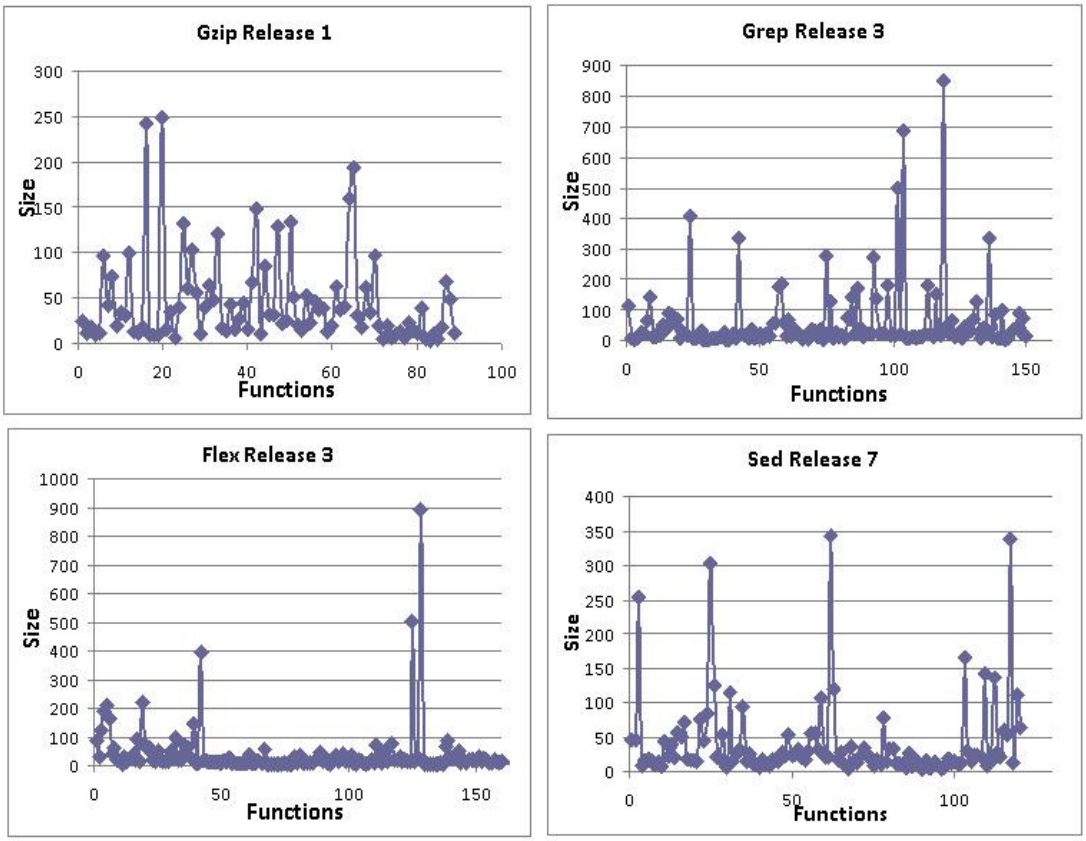


Figure 13: Functions vs. size graph of randomly selected releases of the UNIX utilities (X-axis shows labeled by numbers instead of names and Y-axis shows the size of each function).

2.6.3 Rules of Decision Tree in Understanding Fault Proneness of Faulty Functions

Decision tree model actually generates rules from the independent variables in a training dataset and use those rules to predict a dependant variable (see Section 2.5). In this section, we show examples of decision trees (i.e., rules) for randomly selected functions. These rules are useful in understanding why a particular function could be faulty and

which unique execution paths mostly lead to that faulty function. For example, if a particular function is found faulty in a large number of traces and due to different faults, then a programmer can analyze these rules to find out which execution paths are causing that function to be faulty and perform extensive testing on the functions of those paths.

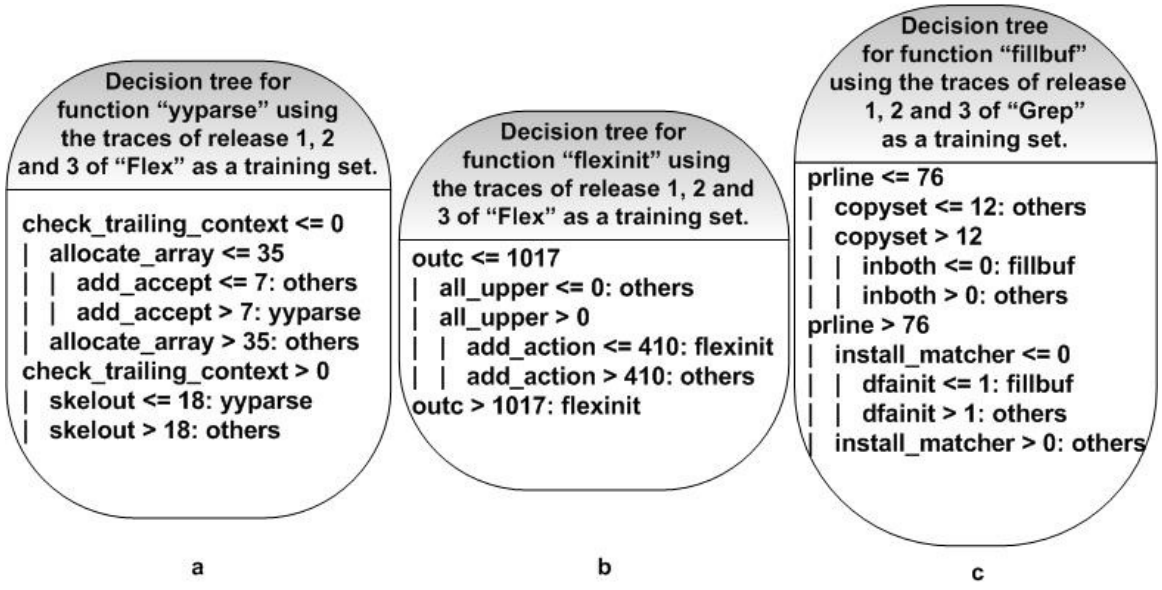


Figure 14: Decision tree models for faulty functions of Flex and grep program.

In Figure 14 ('a', 'b', and 'c'), we show the three decision tree models, generated using the one-against-all approach, for functions of the Flex and Grep program. For example, in Figure 14 part 'a', the rules are read as if occurrence of the function "check_trailing_context" is less than equal to zero, and occurrence of allocate_array is less than equal to 35, and "add_accept" is greater than 7, then the faulty function is "yyparse". In a similar manner, other rules in part 'a', 'b' and 'c' of Figure 14 can be analyzed. Thus, these rules actually provide an abstraction of a collection of faulty traces belonging to faulty functions, and provide a succinct human-readable view of suspicious function-calls (out of many function-calls in traces) related to a faulty function.

2.6.4 Entry Exit Events in Traces

In Figure 2, we showed an example execution trace having "function entry" and "function exit" events. However, we discovered that "function entry" or "function exit" events, by themselves, are adequate to predict the fault origin. Their combined use does

not improve the accuracy. In order to statistically validate this we conducted the Wilcoxon signed rank test (Marques de Sá, 2003) on the accuracy of classification obtained by using both “function entry and exit” and only “function entry or exits” from a trace. We again set the alpha level (or level of significance) to 0.05. We stated the null hypothesis as the mean difference of accuracy between the accuracies of classification obtained using “function entry and exit” and only “function entry/exit” is zero.

Table 7: Classification accuracy for “function entry and exit” and “function entry or exit” (in percentage) using F007.

Program	Episode Rule	Entry and Exits	Entry or Exits
Grep (R3)	1	100.0	95.546
	(S/P) 2	99.190	99.595
	(S) 3	99.595	99.190
	(P) 3	99.190	98.785
Sed (R3)	1	89.361	89.361
	(S/P) 2	92.908	95.745
	(S) 3	92.908	95.035
	(P) 3	92.908	94.326
Flex (R3)	1	58.840	58.563
	(S/P) 2	63.260	61.050
	(S) 3	60.773	61.326
	(P) 3	60.773	61.602
Schedule	1	73.248	74.267
	(S/P) 2	65.350	65.478
	(S) 3	65.605	66.930
	(P) 3	66.369	66.667
Schedule2	1	60.363	60.363
	(S/P) 2	68	65.0909
	(S) 3	63.272	60
	(P) 3	59.272	62.909
Tcas	1	73.481	73.481
	(S/P) 2	73.873	73.546
	(S) 3	73.481	73.807
	(P) 3	73.481	72.045
Tot_info	1	68.842	68.842
	(S/P) 2	68.947	66.831
	(S) 3	67.819	65.831
	(P) 3	63.207	60.244

Table 7 shows the accuracy of classification for function "entry and exit” and function "entry or exit” for all types of episode rules, obtained using window width win(w)=3 for

randomly selected programs only (to save space). In Table 7, for function “entry or exits”, we have randomly selected for some programs only function “entry” symbol and for some programs only function “exit” symbols. This is because using only function “entry” or only function “exit” yield similar number of episodes and accuracy. However, the use of both function “entry and exit” together generates twice as many episodes. The accuracy of function “entry and exit” is also shown in Table 7. All other programs not shown in Table 7 also yield similar results. In Table 7, column “episode rule” includes both the length of episode and type of episode rule (S= Serial, P = Parallel). Finally, the results in Table 7 are obtained by using the confidence values for the function “entry and exits” and “entry or exits”.

The Wilcoxon signed rank test on Table 7 with 48 observations (on 12 programs) yielded z=1.022, (two tailed) p=0.307. This confirms that the null hypothesis cannot be rejected, and there is no significance difference between the classification accuracy of function “entry and exit” and function “entry/exit”. The use of episode rules in the Wilcoxon test also verifies that the patterns of function-calls (higher length episode rules) on function “entry and exits” do not yield better results too. It should be noted that: episodes generated using function “entry and exit” were twice as many as only function “entry or exit”; and episodes generated using function “entry and exit” took twice as much time for extraction from traces as only function “entry or exit”. This not only reduces the processing time for fault discovery to half, but also reduces the space to store traces to half. For example, processing time for traces of release 3 of the Flex program using F007 was: (a) 24.399 minutes for both function “entry and exit”; and (b) 12.584 minutes for only function “exit”. In order to demonstrate, we also measured the size of a trace of the test case “t534” of the fault “F_AA_2” of release 3 of the Flex program. It was found to be 3060 KB with both function “entry and exit”, 1591 KB with only function “exit”, and only 1632 KB with only function “entry”.

This finding also implies that the overhead on the deployed systems, will be half of a normal (function-call) failure trace if only function “entry” or function “exit” is collected for software—reducing the size of a trace to half as well. Thus, in all the results we have shown in previous sections, we have either used only function “entry” or function “exit”

symbols, and discarded the other one—either entry or exit can be discarded. This discovery of function “entry” or “exit” is another distinguishing factor of F007 from previous techniques.

Dallmeier et al. (2005) showed on the SPEC JVM 98 Java programs suite (543 class files, total size 1.48 MB) that the time taken by instrumented software run is almost two orders of magnitude higher than a normal run. Though, tools, environments, and instrumentation methods differ and this can not be generalized, but in any case the time will be reduced to approximately half if only function “entry or exits" are collected. This also means that the trace of the same size could contain two times more information. This is significant for the deployed systems when sometimes a fault cannot be captured in a trace due to its size limitation, for example, if trace-collection is started by a user well before the appearance of failure. During our analysis of a large commercial program, shown in the next section, we found that sizes of some function-call level traces were in several GBs. This discovery would certainly be beneficial in reducing the size and overhead of trace collection to half for such large commercial programs.

2.7 Case Study on a Large Commercial Application

The programs shown in Table 4 ranged from small to medium in sizes and all of them -- except Space -- had hand-seeded faults. Also, apart from the faulty functions in the Siemens suite, all other programs had single faulty function per fault. In this section, we validate F007 on a large-scale commercial application (of size over 20 MLOC) deployed in the field for more than 20 years, have millions of users, developed by several thousands of software engineers over the years, and have many field faults across many functions and components. The characteristics of this software application are shown in Table 8.

Table 8 first shows the general static characteristic of the software application: 82% rediscoveries of the field faults were observed by us in a sample of fault (defect) records (bug/defect records) for a few recent releases; the failed traces, faulty components and faulty functions are for three releases; the last row shows total distinct faulty functions and components for all the three releases.

Table 8: Characteristics of the commercial application under study.

20+ million LOC, 300+ components, approx. 200 K+ functions, and 82% rediscoveries of field faults.			
	# Failed Traces	# Faulty Comp.	# Faulty Func.
Release 1	269	52	65
Release 2	337	35	47
Release 3	99	30	31
Total Distinct Faults (Union)		65	103

We collected failed traces, quantified in Table 8, from the historical trace repository. The average size of a trace in our collection was 50 MB, and often the size of a trace reached several gigabytes. Due to their large sizes, traces for this commercial software are not kept in its repository for a long time and are purged soon after the resolution of the problem. This inhibited us from collecting traces for all the faults. Thus, we collected the failed (field) traces (of the faults) present in the repository for the recent three releases during a period of two years (2007-2009).

In this section, first, in Section 2.7.1, we explain the data collection process for the application. In Section 2.7.2, we discuss richer contents of the function-call level traces of the application, and different heuristics that we used to evaluate F007 on this large application. Afterwards, from Section 2.7.3 to Section 2.7.5, we explain the evaluation of different heuristics using F007. Finally in Section 2.7.6, we show the results of the identification of the faulty functions and faulty components on different releases of this software application.

2.7.1 Data Collection

In order to execute F007 on this large software application, we collected the required data from different sources of this application, as follows:

Step 1: First, we collected execution traces from a "customer-trace" repository containing the execution traces of the software faults reported by customers from the field. These (failed) traces were captured at the time of the occurrence of the faults at the customers' sites, or sometimes reproduced in the lab based on the customers' descriptions.

Step 2: Second, we extracted the program analysis record for each fault from the repository. The program analysis record contained the faulty component, problem resolution, reference to the source code changes, and other related information.

Step 3: Third, we extracted the functions[12], using the references obtained for the faults in Step 2, from the version control repository. These functions were changed because of the faults, and we considered them faulty functions corresponding to the faults. Though not all the functions changed are faulty, no explicit records of the faulty functions are kept for this large industrial application.

Step 4: Fourth, we grouped together all the faulty functions of different faults under one name, if one or more faulty functions for one fault matched the faulty functions of another fault. For example, the faulty functions "foo1", "foo2" for the fault "F1" were grouped with "foo1", "foo3", "foo2" of the fault "F2" as "Group1". The reason is that a fault could occur because of any of these functions and they should be simultaneously identified. In Table 8, "Faulty Func." column shows the number of faulty functions after forming the groups (i.e., the number of groups of faulty functions).

Step 5: Finally, if a fault is found faulty in multiple components we also grouped them together under one distinct name as in Step 4. In Table 8, "faulty comp." column also shows the number of faulty components after forming the groups.

2.7.2 Executing F007 using different heuristics

The next step after the collection of data is to execute F007 on the data. During the analysis of data, we observed that the function-call level traces of this software are richer (in terms of new entities) than what we used for other subject programs (e.g., see Figure 2). An example of execution traces, captured by maintainers, for this software application is shown in Figure 15, which has, in addition, probe points and error codes compared to only "function entry" and "function exits" of Figure 1. The probe points are the specific

[12] A faulty function name is extracted with its scope (e.g., namespace, file), because two functions in different namespaces can be same.

locations within a function that are executed during a software run. The error codes represent the exceptions thrown during a software run.

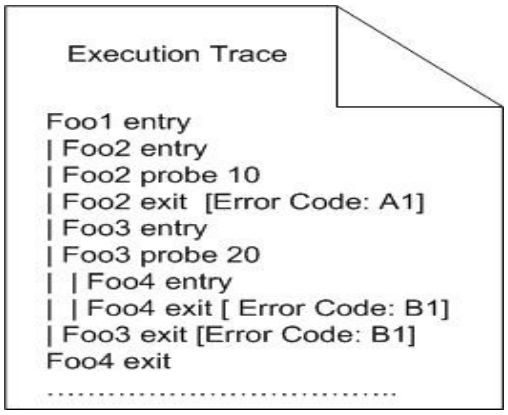


Figure 15: Example execution trace of the large commercial software (with names obfuscated for privacy reasons).

In Section 2.6.4, we showed that the use of only one of "function exit" or "function entry" is necessary to predict the faulty functions. Considering this fact for a trace in Figure 15 as well -- since the traces in Figure 2 and Figure 15 are similar-- we further investigated the properties of the additional characteristics of the trace shown in Figure 15 using different heuristics. We trained F007 on: (a) "function exits", "function probe points", and "function exit with error code"[13]; (b) only "function exits" without "error codes" and "probe points"; and (c) only those "function exits" which have large variations in occurrences across the execution traces.

2.7.3 Evaluating Heuristic 'a'

The results of the identification of faulty functions in the failed traces of this software, using F007 with the three heuristics (defined in Section 2.7.2), are shown in Table 9. The results in Table 9 are obtained by training F007 on 25% traces of release 2 and testing on 75% traces of release 2. We chose release 2 to show the results because it has the largest number of failed traces. All other releases also exhibit similar results.

[13] Each "function exit", "function probe", and "function exit with error code" is a length 1 episode.

Table 9: Identifying the faulty functions in the failed traces of a large software system by reviewing less than 1 % of the code (functions).

# of Functions Reviewed	% of Code Review	Heuristics						
		ALL	EXITS	SD> 10	SD > 20	SD > 100	SD > 200	SD > 400
1	0.1	52.0	52.4	52.8	52.8	56.0	54.6	54.6
2	0.2	55.1	52.8	53.3	53.3	56.8	56.0	55.5
3	0.3	56.8	57.7	58.2	58.2	61.3	60.4	59.1
4	0.4	62.7	58.6	58.6	58.6	62.2	61.3	60.4
5	0.5	64	64.8	64.8	64.8	64.4	63.5	62.6
6	0.6	73.7	67.1	67.1	67.1	68.4	68.4	68.0
7	0.7	74.6	75.1	75.1	75.1	78.6	78.6	78.2
8	0.8	74.6	75.5	75.5	75.6	79.1	79.1	78.6
1000	100	100	100	100	100	100	100	100
		Episodes for Heuristics						
		17331	10481	6999	6190	3611	2686	1892

For example, column "ALL" in Table 9 shows the results of F007 with heuristic 'a' for release 2. First cell in column "ALL" shows that the faulty functions in 52% of the traces in a test set of release 2 is identified by reviewing only the first function in the list. The list of faulty functions is generated using F007 from a training set of release 2. Similarly, second cell shows that faulty functions in 55.1% of the failed traces are identified by reviewing only the first two functions in the list. This goes on up till 74.6% of the traces are diagnosed till the review of eight functions in the F007's list of faulty function. F007 was unable to identify the faulty functions in the remaining 25.4% of the traces using heuristic 'a' ('ALL').

In this software, there were more than 200,000 functions, but we assumed[14] that approximately 1000 functions would be required to review for the rest of the 25.4% (cumulative 100%) of traces (see the 9th row in Table 9). The code review is measured in Table 9 is measured by dividing functions reviewed by 1000 functions. The code review, in Table 9, in terms of functions is less than 1% up till the 8th function which is very

[14] On discussion with developers, it is more than 1000 but we considered the minimum number of functions that a developer may consider using experience and contextual information.

significant for this software. Finally, the last row of Table 9 shows the number of episodes generated for the heuristic using the sample traces we studied; e.g., for heuristics ‘a’ the number of episodes of length 1 (i.e., sngle function-calls) was 17331.

2.7.4 Evaluating the Heuristic ‘b’

In Table 9, column “EXITS” shows the results of F007 with heuristic ‘b’. For this, we trained and tested F007 only on the “function exit” excluding “error codes”. Also, we did not use the function-calls with the “probe points”. It can be observed that accuracy of identifying faulty functions is similar for only “function exits” (heuristic ‘b’), and “function exits with probe points and error codes” (heuristic ‘a’). A Wilcoxon Signed Rank test (Marques de Sá, 2003) between the results of heuristic ‘a’ and heuristic ‘b’ show that no significant difference exists because: $z=0.420$, observations = 9 and two tail $p=0.674 > 0.05$.

Note that in heuristic ‘b’ we removed “error codes” (i.e., the type of error); however, interestingly enough the results were similar to heuristic ‘a’. *This again shows that different faults in the same function occur with similar sequence of function-calls irrespective of the type of the fault within a function* (see Section 2.6.2). Also, note that the number of episodes for heuristic ‘b’ is 10481 compared to 17331 episodes of heuristic ‘a’. This means heuristic ‘b’ is better than heuristic ‘a’ because the smaller number of episodes consume smaller amount of memory and results in efficient decision tree generation.

2.7.5 Evaluating the Heuristic ‘c’

During our experiments on this commercial application, we observed that a large number of length 1 episodes (“function exits”) occur with a small variation in their occurrences across traces because: (a) they occur in very few traces; or (b) they occur with a similar number of occurrences across traces. If episodes occur with the same (or very similar) occurrences across traces of different faulty functions then they do not contribute much in classifying faulty functions for a trace using the decision tree. Similarly, if episodes occur in very few traces of the same faulty function then they also do not help in distinguishing the same faulty functions in different failed traces. In order to determine whether

episodes with small variations in traces of this commercial application affect the accuracy of predicting faulty functions, we employed heuristic ‘c’.

In heuristic ‘b’ we already determined “function exits” are sufficient for predicting faulty functions; therefore, in heuristic ‘c’ we use only “function exits” as episodes of length 1. In heuristic ‘c’, we set different thresholds for standard deviations of occurrences of (length 1) episodes (“function exits”). We then trained F007 on the episodes with standard deviations higher than the set threshold. The objective here is to remove unnecessary episodes such that: the efficient decision tree can be generated with smaller memory consumption; the accuracy of identifying faulty functions remains the same as heuristic ‘b’ (“EXITS”); and the failed traces should not get excluded.

We set thresholds of standard deviation of occurrences from 10 to 500 with steps of 10, and then trained F007 on the episodes of length 1 with standard deviations of occurrences higher than each of the threshold values. We show the results in Table 9 for selected threshold values to avoid cluttering the text. The results in Table 9 are obtained by training F007 on 25% traces of release 2 and testing on 75% traces of release 2—similar to heuristic ‘a’ and ‘b’. In Table 9, the columns starting with “SD” (Standard Deviation) show different thresholds for standard deviation of occurrences of episodes. For example, first cell in the column “SD > 10” shows that the faulty functions in 52.8% of the failed traces were identified by training and testing F007 on episodes with standard deviation of occurrences greater than a threshold of 10.

We stopped at the threshold of 400 because beyond this value a number of failed traces started to get excluded from the train and the test set. It can be observed from Table 2 that the accuracy remains similar between heuristic ‘b’ “EXITS” and different thresholds of standard deviation in heuristic ‘c’. The results obtained from standard deviation of 400 has minimum number of episodes, we consider it as the best case for heuristic ‘c’. A Wilcoxon Signed Rank test with 9 observations between the results of heuristic ‘b’ (“EXITS”) and the results of heuristic ‘c’ with “SD > 400” yielded: $z=0.630$ and $p=0.529$; that is, no significant difference exists as $p > 0.05$.

Recall from the beginning of this section that our intuition was to remove the episodes with small variations; however, the standard deviation of 400 occurrences would seem quite high. If we compare this standard deviation of 400 occurrences with the maximum standard deviation of 358,945.53 occurrences in the traces of this large software, then it is quite a small variation. Thus, episodes (or simply functions) with large variations are fewer and contribute efficiently in the decision tree model in identification of faulty functions in the traces.

Similarly, we evaluated the three heuristics on two other releases (release 1 and 3) and. the accuracy of identifying the faulty functions remains similar. Heuristic 'c' for release 1 and 3 also resulted in similar results. Thus, we considered "SD > 400" as the threshold point. Our results in the rest of the section are obtained using episodes with standard deviation of occurrence higher than the threshold of 400.

In terms of other programs used in this study, exclusion of function-calls with small variance resulted in the exclusion of failed traces. This is because there were only a few hundred function-calls in other programs compared to 17000 function-calls (from the sample of traces we used) of the large software application. However, using all the function-calls for the larger program will also result in the same accuracy, as shown here, but it will consume more space and memory.

2.7.6 Identifying the Faulty Functions and Components across Releases

In Figure 16, we show the results of F007 on three releases by using 25% training set and 75% test set. It can be observed that on average the faulty functions in 70% of the failed traces are successfully identified by F007 for each of the releases by reviewing less than 1% of the code (functions). In the rest of the 30% cases some of the faulty functions occurred only once (one trace). So these functions were not identified at all by F007 for the sample of traces we used. However, there were 82% rediscoveries of the faults in the database and the traces were not kept for a long time in the repository of this commercial software due to their large sizes. This is why we have a few faulty functions found only

in one trace. F007 stores traces in its database in the form of common functions (episodes); thereby, reducing the storage overhead required to store traces in the raw form. Thus, actual raw traces can also be preserved for a long time by F007.

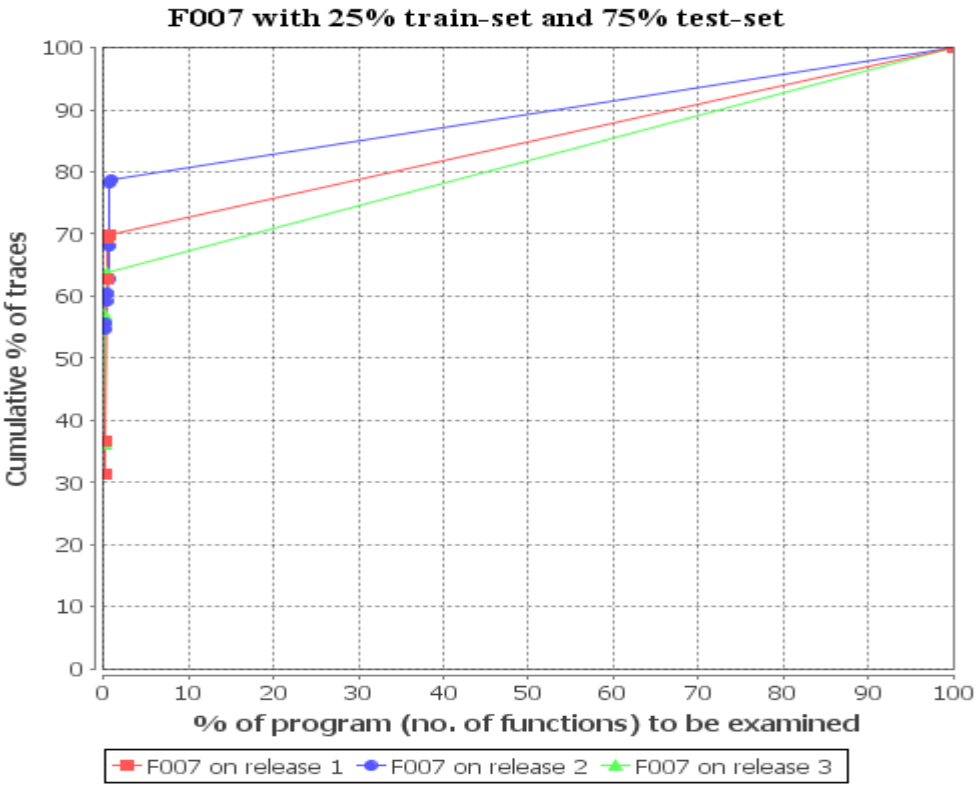


Figure 16: F007 on three releases of a large commercial application.

In Figure 17, we show the results for identification of the faulty functions in release 2 by using release 1 as a train-set for F007. By using traces of release 1 we were able to only identify faulty functions in 35% of the failed traces of release 2 on the review of 3% or less of the code. This is because not all of the faulty functions found in release 2 were present in the training set of release 1. However, on using 10% traces of release 2 with the traces of release 1 approximately 80% of the faulty functions were successfully identified. Similarly, in Figure 18, we have used the traces of both release 1 and release 2 to identify the faulty functions in the traces of release 3. Figure 18 shows that the faulty functions in about 60% of the traces were identified by using only the traces of release 1 and release 2.

In our experiments in Section 2.6.2 and in this section (Figure 17), interestingly, we observed that in the first few releases there are fewer common faulty functions than in the subsequent releases (e.g., Figure 18). It could be due to the sample of data that we used for experiments did not contain common faulty functions in the failed traces. It could also mean that as the software gets stable through releases, the number of faulty functions become similar. Nonetheless, if 50-90% of the field failures are rediscoveries of the same fault then by using just 10% of the failed traces of current release we can still identify the majority of the faulty functions. Also, in the case of earlier releases, the accuracy of F007 can be improved by using in-house failed traces because we have observed that: fault in the same function occurs with similar function-call sequences; and there is an overlap among origin of in-house and field faults according to our own study on a very large software system (Gittens et al., 2005).

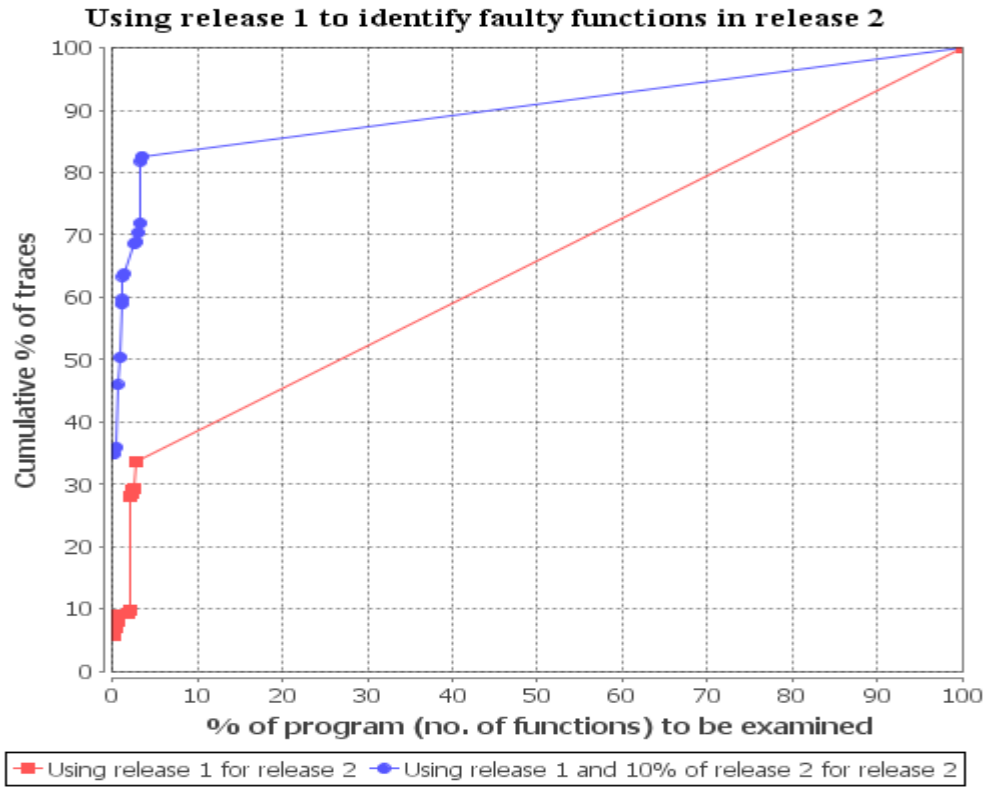


Figure 17: Identifying the faulty functions across releases.

In the large commercial software application it would be worthwhile to point out faults at a higher level of granularity too, such as components. A large system actually contains so

many components that it makes the component level a useful abstraction for maintainers to locate bugs in functions, files, and statements. This could aid maintainers in correctly diagnosing the fault origin. For example, maintainers can use their experience to decide which combination of faulty functions and faulty components from F007's predicted list would lead them to the correct fault origin.

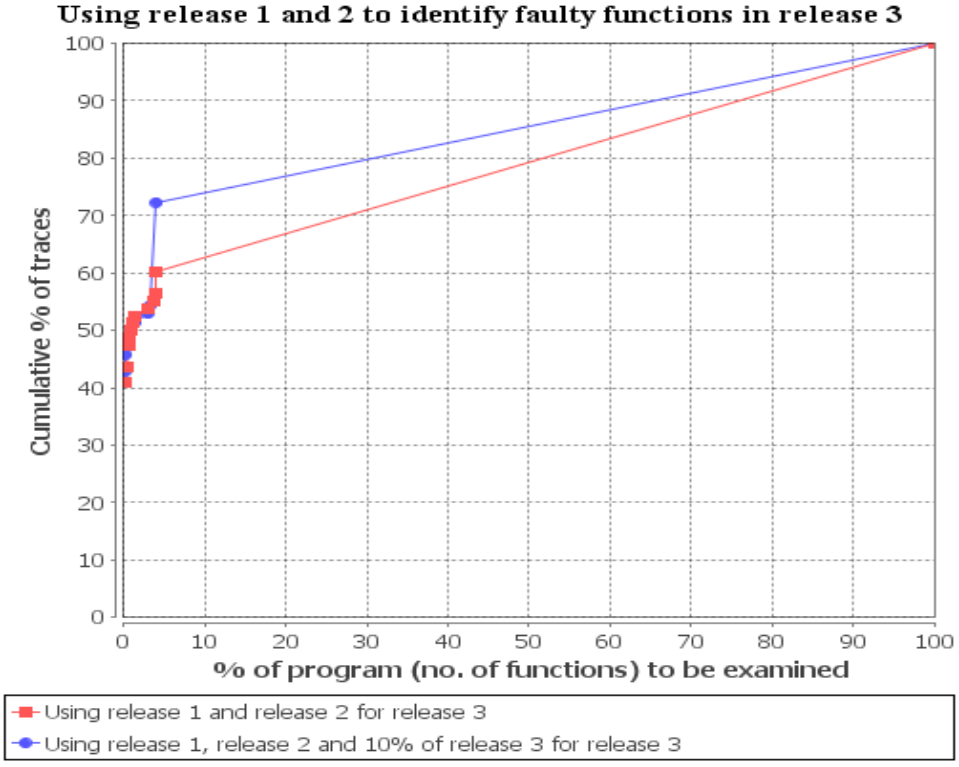


Figure 18: Identifying faulty functions across releases.

In Figure 19, we show the accuracy of F007 in identifying the faulty components in the field traces. Here, we used 300 components as the total number of components to measure the code review in terms of components. We first used release 1 to identify the faulty components in release 2. It can be seen in Figure 19 for the series "using release 1 for release 2" that faulty components in approximately 50% of the failed traces were diagnosed correctly by reviewing 8% of the program (i.e., components in this case). Similarly, second series "using release 1 and 10% of release 2 for release 2" shows that faulty components in approximately 90% of the failed traces of releases 2 can be correctly identified on reviewing approximately 8% of the code. This identification of

faulty components was done by using only 10% of the failed traces of release 2 and the traces of release 1 as a training set, and the remaining 90% of the traces of release 2 as a test set. Finally, following the similar approach, Figure 19 also shows that faulty components in 90% of failed traces of release 3 were identified by using only failed traces of release 1 and release 2. This identification is done by just reviewing approximately 8% of the code (components).

In Figure 19, we have not shown the results for release 3 from the combination of traces of release 3, release 2 and release 1. This is because 90% of the components in release 3 were already identified using the traces of release 1 and 2 on the review of approximately 8% or less of the code (components).

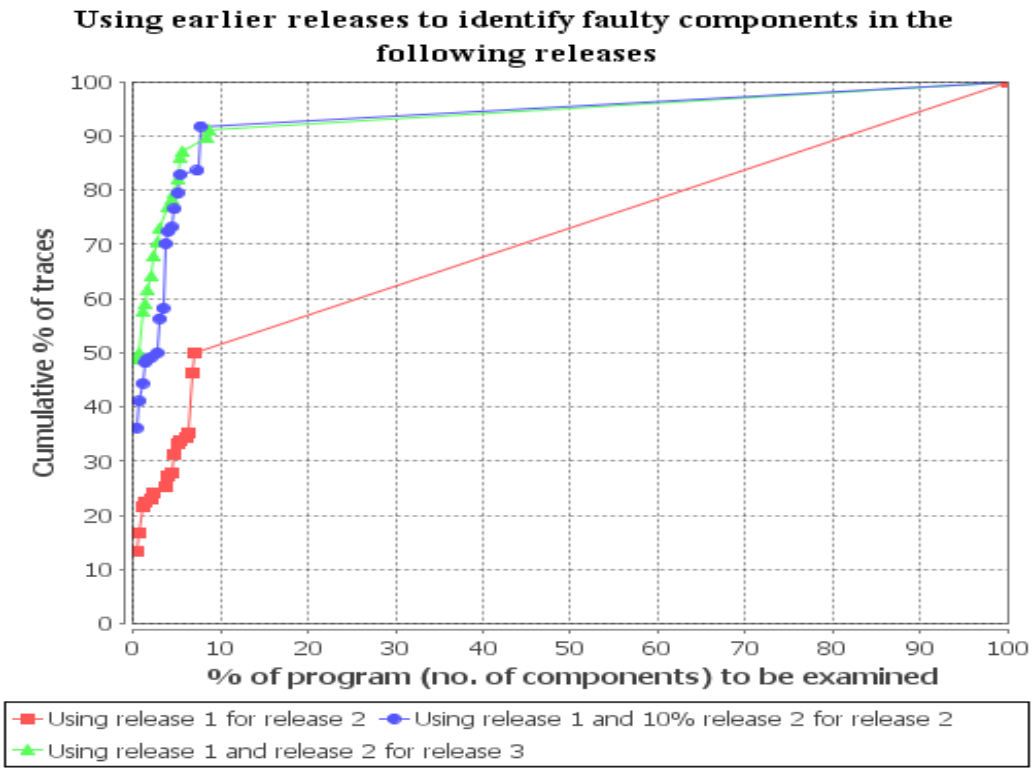


Figure 19: Identifying faulty components across releases (a total of 300 components make 100% program in this figure).

In the results of this section, we have only shown the results in terms of number of functions or components reviewed (program). For this commercial software, we could

not get access to the actual source code to count the number of statements of functions or components. Doing so would help in finer-grained evaluation in terms of the number of statements reviewed for each function or component (similar to what we have shown in Section 2.6.2.3. However, as we mentioned in the earlier section (Section 2.6.2.3, maintainers do not review all the statements of every component or function to identify a fault; they use their experience and context to focus only on the few relevant statements. Thus, further finer-grained evaluation would be based on the subjective judgment.

2.8 Executing F007 across Releases: Revisiting Example Execution

In Section 2.6.2 and Section 2.7 we showed that F007 can identify faulty functions in failed traces of the current release of software by using failed traces of previous releases. However, the example execution of F007 shown in Section 2.5 described an example of predicting faulty functions in the same release. In the case of predicting faulty functions across releases, the following issues need to be considered prior to training F007:

1. In a new release there could be newer faulty functions that are not present in the current release. In order to build the decision tree model we need to add those newer functions in the training set of failed traces of previous release with the confidence of 0. This is because they do not exist in the previous release and may or may not occur in the traces of the current release. For example, in the “Flex” program function (or episode of length 1) “format_pinpoint_message” did not exist in the failed traces of release 1 and release 2, and was found in only in the failed traces of release 3.

2. In Section 2.7 we mentioned that in the large software application, we can identify faulty functions by using function-calls (episodes of length 1) of higher variances. Therefore, in the case of the identification of the faulty functions in newly failed traces in the same release only those functions should be used which were identified as having higher variances in the historical collection of the failed traces. In a new release, in-house failed traces can be added to the historical

collection of failed traces of previous releases to identify functions with higher variances. Later, as failed traces from the field arrive, only those functions identified with higher variances should be considered for discovering faulty functions in the trace. Note that this process is carried out offline, does not affect software in the field, is not always required to be done, and can be repeated at regular intervals when there are enough (e.g., 5% of the historical collection) new failed traces.

In short any one of the two procedures: (1) adding new functions or (2) using the functions with higher variances, can be used. This is because both of them yield the same results (see Table 9).

2.9 Summary of the Results

Our results in Section 2.6 on the Siemens suite (Hutchins et al., 1994), the Space program (Do et al., 2005), the four UNIX utilities (i.e., Flex, Grep, Gzip and Sed) (Do et al., 2005), and on the commercial application (200K functions) in Section 2.7 show that:

- Patterns (episode rules) of function-calls do not yield better results than the single function-calls when used with the decision tree in identifying faulty functions (see Section 2.6.1).

- When function-call level execution traces are used then only "function entry" or only "function exits" are adequate for discovering faulty functions (see Section 2.6.4). This discovery implies that the size of the trace, and run time overhead of the function-call level trace collection could be halved. This discovery also applies to the execution traces of larger software application with richer semantics (e.g., exceptions thrown)—i.e., events other than function "entry" or "exit" can be ignored However, the thrown-exceptions may be useful in understanding the type of fault (see Section 2.7.2)

- Faults in the same function occur with the similar sequence of function-calls because F007 can identify the same faulty functions (with different types of

faults) in the failed traces, if only the failed traces of at least one fault in the same function are known (see Section 2.6.2. and Section 2.7).

- F007 can accurately identify faulty functions in traces of a current release by using failed traces of previous releases, provided that the faulty functions are present in the traces of previous releases (see Section 2.6.2. and Section 2.7.6).

- F007 can also identify faulty functions in the same release by using only few (e.g., 10%) failed traces. If 50-90% of the field failures are rediscoveries than this can be highly beneficial; e.g., using few earlier traces of at least one fault in a function, the same faulty function in new upcoming field traces can be identified (see Section 2.6.2. and Section 2.7.6).

- F007 yields different results with different empirical settings but, in general, F007 can identify faulty functions in approximately 65-90% of the failed traces on the review of first few (e.g.,1-4) functions (see Section 2.6.2. and Section 2.7.6).

- In the commercial software system, F007 can identify faulty functions using only single function-calls (length 1 episodes; see Section 2.7.2) of higher variance in the failed traces. This is because in the large software application many function-calls (e.g., system calls) occur in few traces, or occur in small variations across traces: ignoring such function-calls, in large software system, did not affect the accuracy of fault-origin discovery, and did not exclude any traces (see Section 2.7.5). This reduces many unnecessary function-calls in large software, generates efficient decision tree by consuming less memory and space, and results in the same accuracy as using all the function-calls (see Section 2.7.5). In terms of other programs used in this study, exclusion of function-calls with small variance resulted in the exclusion of failed traces. This is because there were only a few hundred function-calls in other programs compared to 17,000 function-calls (from the sample of traces we used) of the large software application (see Section 2.7.5). Thus, the results show that for medium to small programs all the function-calls can be used to discover faulty functions; whereas, for large programs function-calls of higher variances or all the function-calls can be used (see Section 2.7.5).

We used F007 on a variety of software applications (see Table 4 and Table 8), with hand seeded and real faults, having research and commercial applications. In all the cases, the results were similar and show the significance of F007. In terms of the UNIX utilities, faults were hand seeded but they were added in the actual changes in the sources code from one release to another. This makes them quite realistic (Do et al., 2005) because changes in a program are made due to faults or modification for new functionality. This is one of the reasons why the same 20% of the source code is responsible for 80% of the faults. F007 can accurately identify the faults in the same area of code (i.e., functions or components). In the case of new functionality (e.g., new functions and components) F007 would still work if it is trained on a small percentage of new field failed traces (or in-house traces) and will identify 50-90% rediscoveries.

2.10 Comparison with Contemporary Techniques

In, Figure 20 we juxtapose the results of F007 and those of other fault localization techniques -- using the Siemens suite. These other techniques include Frequent Pattern Mining (FP) (Di Fatta et al., 2006) and Tarantula (Jones and Harrold, 2005) on function coverage taken from the work of Di Fatta et al. (2006). Tarantula was actually proposed for statement coverage by Jones and Harrold (2005). In Figure 20, Y-axis (named as axis 1) for the FP and the Tarantula is measured in the percentage of versions. A fault was equivalent to one version in the Siemens suite. Each version contained many passing and failing traces. FP and Tarantula actually discovered a faulty function containing a fault, by using passing traces and failing traces pertaining to that fault (or version).

In Figure 20, we also show the performance of F007 on the Siemens suite; however, the Y-axis (axis 2) for F007 is calibrated in the number of failed traces for all versions (faults) of the Siemens suite. This means that F007 can discover the faulty functions in a single trace using the previous collection of (only) “failed” traces for the same or different faults in the same function. F007, unlike FP and Tarantula, does not require a collection of “passing” traces and “failing” traces related to the same fault in a faulty function to discover that faulty function. However, F007 still requires an initial collection of labeled traces with known faulty functions (i.e., the knowledge of at least one fault for the function) to discover the faulty functions in new traces. (See Section 2.3 where we

describe how an initial set of traces can be built from in-house traces and subsequently can be evolved from field traces, and how F007 can be trained on the evolved set of traces.)

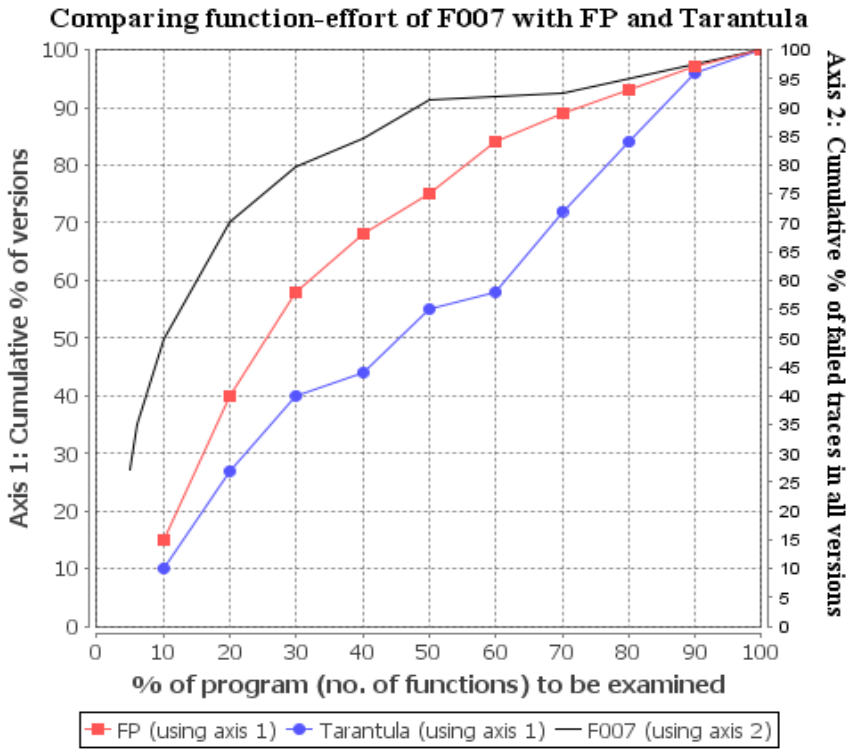


Figure 20: Comparing Frequent Pattern Mining (FP) using function sequences and Tarantula on function coverage against F007.

Thus, F007 is useful for deployed software where a large number of faults are rediscoveries originating from a small percentage of code. It is also useful when it is not feasible to collect many passing traces and failing traces for a fault from the field, or when only the failed traces are gathered for economic reasons. FP and Tarantula are suited primarily for in-house testing where pass-fail traces are readily accessible for a fault, but they are not suitable when only limited failing traces are available from the field. Thus, while F007 is related to FP and Tarantula, it is not directly comparable because F007 is suited for field testing and FP and Tarantula are suitable for in-house testing. Similarly, other techniques mentioned in Section 2.2.1 (e.g., discovering faulty statements using statement coverage (Jones and Harrold, 2005; Wong et al., 2006; Wong

et al., 2007; Zhang et al., 2009); statistical debugging (Chilimbi et al., 2009; Liu and Han, 2006; Liu et al., 2005; Zheng et al., 2004) also have the same major differences with F007 as do FP and Tarantula in Figure 20. A similar comparison of F007 in terms of effort in statements is made against the statement-level techniques, effective fault localization using code coverage (EFL) (Wong et al., 2007) and Tarantula on statement coverage (Jones and Harrold, 2005), in Figure 21. Again, the same differences exist between F007 and EFL and Tarantula, and the results are not directly comparable for the same reasons as mentioned before for Figure 20 (FP and Tarantula on function coverage). The statement effort for F007 would only improve as it was the pessimistic approach (see Section 2.6.2); whereas, the statement effort for EFL and Tarantula, in Figure 21, would not improve further --it is the best case. In Section 2.2 and Table 2, we characterized F007 and the other closely related techniques similar to Tarantula and EFL. There is no direct comparison of F007 against other fault discovery techniques focusing on in-house testing.

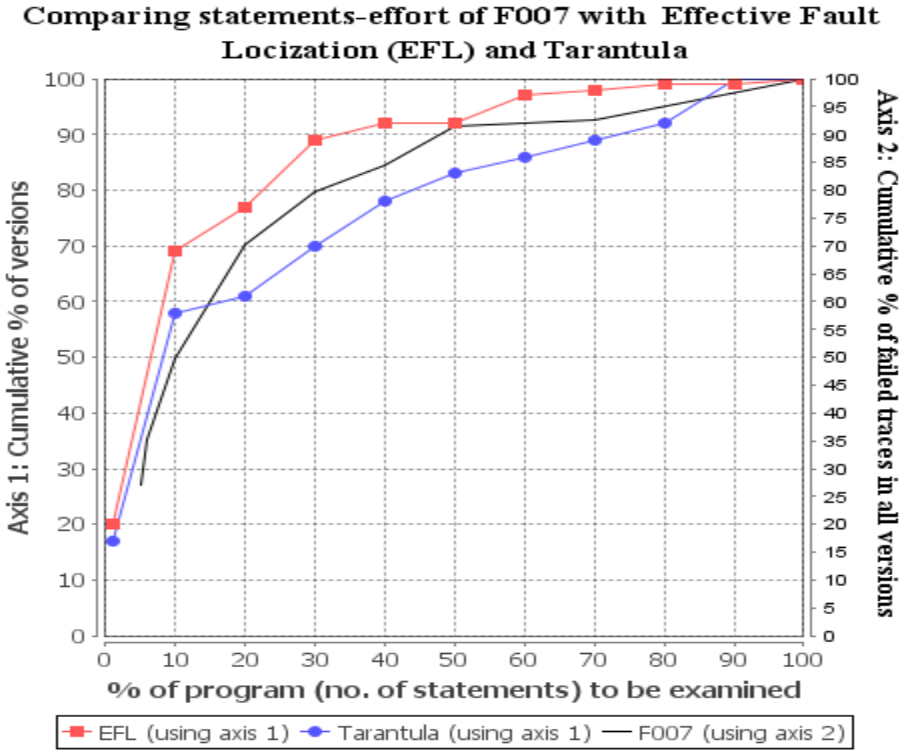


Figure 21: Comparing Effective Fault Localization and Tarantula on statement coverage against the statement-effort of F007.

Table 10: Comparison of related techniques focusing on function-call pattern analysis.

Reference	Pattern Length	Pattern Type	Method	Output
Di Fatta et al. (2006)	2+	Serial	Heuristic	Function
Dallmeier et al. (2005)	2+	Serial	Heuristic	Class
Elbaum et al. (2007)	5	Serial	Heuristic	Pass/fail
Yuan et al. (2006)	1	Serial	Classifier	Config. Cause
F007	1	Serial, Parallel, and Hybrid	Classifier	Function

In Section 2.6.1 we showed that only single function-calls (episodes of length 1) are sufficient to discover faulty functions in failed traces. In Table 10, we provide a comparison of our findings with those of the related techniques focusing on the use of patterns in fault discovery. Table 10 shows that: (a) the references of the related techniques focusing on the use of patterns; (b) the length of function-call patterns that other researchers found effective in improving accuracy; (c) empirical method employed by researchers (a machine learning classifier or other comparison heuristics); and (d) the output of techniques. The techniques in Table 10 are explained as follows:

- The technique (FP) to detect faulty functions by Di Fatta et al. (2006) and using object-specific sequences to detect faulty classes by Dallmeier et al. (2005) were primarily focused on testing. They (Di Fatta et al., 2006; Dallmeier et al., 2005) compared patterns of functions-calls extracted from passing traces against the patterns from failing traces to detect faulty functions (Di Fatta et al., 2006) or (Java) classes (Dallmeier et al., 2005). They (Di Fatta et al., 2006; Dallmeier et al., 2005) found that patterns of length greater than two function-calls discover faults with 15% to 20% better accuracy than length 1 functions. Di Fatta et al. (2006) experimented on the Siemens suite and Dallmeier experimented on NanoXML (4334-7646 LOC and 16-23 classes).

- Elbaum et al. (2007) found out that patterns of length up to five are useful in deciding when to start the collection of the traces of field failures. They found (Elbaum et al., 2007) that function-call patterns of length up to five improve accuracy by 10% from length 1 function-calls, but the accuracy does not improve

beyond length five. Elbaum et al. (2007) use heuristics such as identification of exceptional function sequences and exceptional frequency ranges to achieve their task on the Pine program (157,245-186,366 LOC and 1558-1785 functions).

- Yuan et al. (2006) use support vector machines (a classification algorithm) to identify root causes of the configuration problems in a Windows XP based system. Due to the large size of the Windows XP, their traces contained about 100,000 system function-calls. Yuan et al. (2006), like F007, found out that patterns of function-calls of higher length do not yield any better accuracy than single function-calls.

- Finally, we evaluated F007 on small to large commercial programs (see Table 4 and Table 8), and found out that when using the decision tree classifier higher length patterns of function-calls do not improve accuracy. Our findings our similar to what Yuan et al. (2006) found when using another classifier. However, Yuan et al. (2006) (including other researchers in Table 10) only extracted serial patterns (of length equivalent to window width); whereas, we have extracted serial, parallel and hybrid patterns (see Section 2.3.1) of different window widths and length sizes--our experiments cover a wide range of patterns. We have also validated our results by conducting statistical tests on many different programs (see Section 2.6.1.1); other researchers' works in Table 10 lack on this front.

Another novel contribution of this paper is that it identified that the use of only function "entry" or only function "exit" is sufficient to discover fault origin (see Section 2.6.4). This discovery helps in reducing the size and overhead of function-call traces to half; e.g., the large program used in our study, in some cases, has traces of about 4GB (44 million function-calls)—such traces can be reduced to half. Also, in the case of the large program (see Section 2.7.5), we remove those functions which had low variances because they occur in few traces or occur in all the traces. Yuan et al. (2006) also performed similar filtering by setting a threshold to remove function-calls occurring rarely in some traces. Interestingly, in Yuan et al. (2006) and in our case the accuracy remains same after removing such function-calls. Thus, this shows that in case of the large programs,

the sizes of function-call traces can be reduced to more than half—if rarely occurring or function-calls with low variances are discarded along with function “entry” or “exit”.

In summary the novel attributes of this paper are: (a) faulty functions in future releases or the same release can be identified by using the traces of at least one fault of the same faulty functions from previous or the same release; (b) different faults in the same function occur with similar function-calls; (c) patterns of function-calls (i.e., serial, parallel, and hybrid) do not improve the accuracy of identification of fault origin—single function-calls are sufficient; (d) only function “entry” or only function “exits” are sufficient to discover the fault origin; and (e) in the large program, the removal of function-calls with similar frequencies do not decrease the accuracy of identification of faulty functions.

2.11 Threats to Validity

In this section, we describe certain threats to the validity of the research results. We classify threats into four groups: conclusion validity, internal validity, construct validity, and external validity.

2.11.1 Conclusion Validity

Conclusion validity is concerned with our ability to draw the correct conclusion about the relations between treatment and outcome of an experiment (Wohlin et al., 2000).

A threat to conclusion validity exists with traces of the number of faults we used to infer the conclusion. In the large software application, in Table 8, we observed 82% rediscoveries of faults in the database, but we were able to collect failed traces of only some of the faults. Similarly, in terms of the UNIX utilities, the failed traces for some faults were not used because of the criteria of using the faults with less than 20% of failed test cases (Do et al., 2005). The sample of failed traces that we collected did not represent all the faults that occurred in the releases of the software applications. This threat is mitigated by the fact that results in the large software application were similar to the results of the Siemens suite (Do et al., 2005; Hutchins et al., 1994), Space (Do et al., 2005) and UNIX utilities (Do et al.,2005). In fact, the accuracy across releases would be

higher if the failed traces of all the faults were used. This is because the decision tree would have had sufficient knowledge of cross-release faulty functions, and resulted in a better accuracy.

The threat to conclusion validity is low because we have evaluated F007 on twelve medium to large programs with several releases, and a large legacy program with three releases. There is thus sufficient evidence for valid conclusions.

2.11.2 Internal Validity

Internal validity is concerned whether the relationship between treatment and outcome is causal, and not due to any confounding factors (Wohlin et al., 2000).

A threat to internal validity exists in the implementation of the algorithms and this technique, since it involved quite a lot of programming. Human errors (e.g., logical errors) are a possibility in the implementation of the algorithms. Though, it was not possible to manually verifiy the output on all the traces for the MINEPI algorithm, we have mitigated this threat, and made our implementation reliable, by manually investigating the outputs on different example traces. For example, in the case of the MINEPI algorithm (Mannila et al., 1997), we manually verified the outputs on different examples.

2.11.3 Construct Validity

Construct validity refers to the extent to which the experiment settings actually reflect the construct under study (Wohlin et al., 2000).

A threat exists in measuring the programmer's effort in discovering faulty functions. Recall, from Section 2.3, 2.5, and 2.6.2, that F007 generates a list of faulty functions for a new failed trace, and the programmer's effort is measured by counting functions (or statements) examined (see Equation 1 and Equation 2). In a ranking based technique, such as F007, it is possible that two or more functions can be listed at the same rank. In such cases, the best case is the first function to be examined is faulty and the worst case is the last function to be examined is faulty. For example, suppose there is one function listed at rank 1, five functions listed at rank 2, and one of the five functions at rank 2 is

faulty. The best case is that the faulty function is the second to be discovered (i.e., one at rank 1 and one at rank 2), whereas the worst case is that the faulty function is the sixth to be discovered. This implies that an incompetent technique will have high best case accuracy (e.g., 90-100% accuracy on examining 1-10% of the program) and low worst case accuracy (e.g., 90-100% accuracy on examining 90-100% of the program), because it will list all the functions as faulty at the same rank.

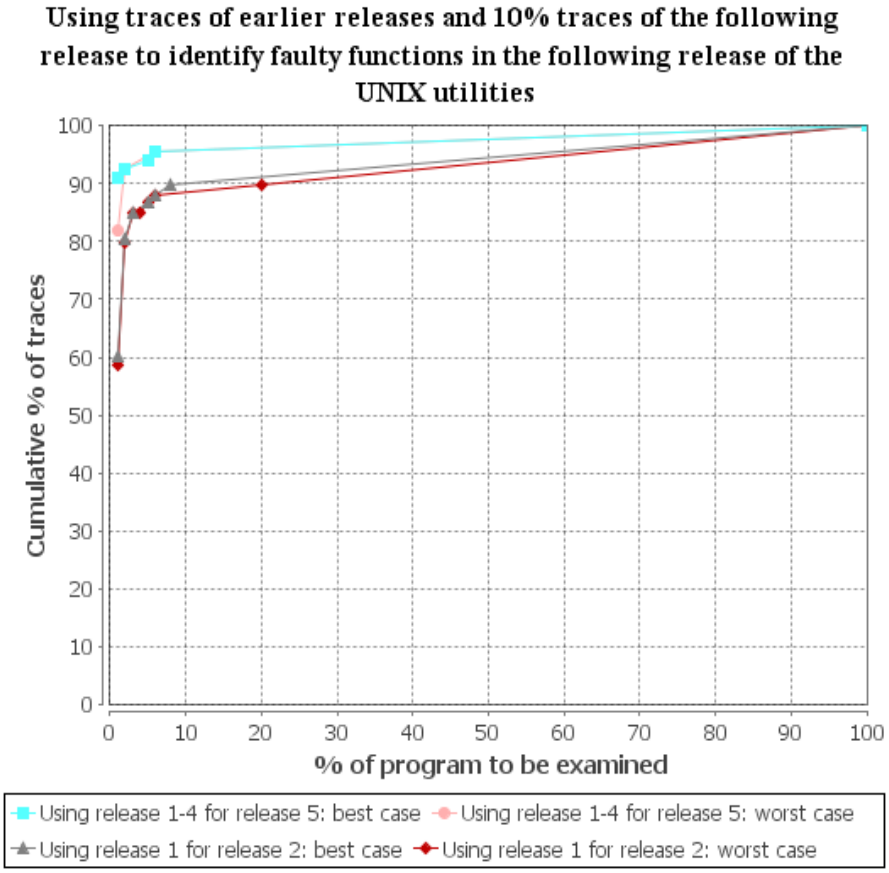


Figure 22: Best and worst case accuracies using F007 for the UNIX utilities.

In our case, the worst and the best case mostly resulted in the same accuracy: in few cases, there were only minor difference between the worst and the best case. For example, Figure 22 shows the examples of the worst and the best case accuracy differences on the UNIX utilities obtained using F007. Thus, in all our results we have shown the best case accuracies because: (a) there was hardly any difference between the worst and the best case; (b) this avoided cluttering of graphs; and (c) related techniques

demonstrate their best cases, a valid comparison could only be made by comparing their best cases with our best case.

Another threat to construct validity exists in measuring the code reviewed by the programmer to identify faulty functions. This measure of programmer's effort was dependent on the faulty traces and their correct mapping to faulty functions. In the case of the large software application, as mentioned in Section 2.7, no record of direct mapping between faulty functions and failed traces was kept. We collected the required data by using the help of different developers, tools and scripts. The process was complex and it could have resulted in discrepancies in the mappings of traces to the faulty functions in some cases. This threat was mitigated by the fact that the results on the very large software application were similar to the results of other software studied. Also, this threat was mitigated by using sufficient number of traces for the large software.

A threat to construct validity exists in the use of failed field traces for fault discovery by F007. Consider, automated failure reporting such as in Mozilla, Net Beans, and Visual Studio. This failure reporting facilitates fault localization by providing contextual information, traces, etc. to the developers. It may be possible that such large number of traces may contain passing traces. In such cases, pass-fail classification techniques (Bowring et al., 2004; Haran et al., 2007) or a technique to collect only function-calls related to the fault (Elbaum et al., 2007) (which are complementary to our work) can be used to classify a trace as passing or failing. However, if a trace is captured at the time of a fault (as it was in the case of the large program; see Section 2.7), then F007 will identify faulty functions in that trace. This is because if the trace is captured at the time of a fault then it would encompass the sequences of function-calls contributing to faulty functions; even if it doesn't contain specific fault conditions (e.g., exception thrown). Thus, F007 can identify faulty functions in such field traces because our results (in Section 2.7) on the large program are not different from other subject programs.

2.11.4 External Validity

External validity refers to the ability to generalize results of an experiment to industrial practice (Wohlin et al., 2000).

If a commercial software application is restructured, then a threat exists when predicting faulty functions across the releases. In the restructuring of software application new functions are added, previous functions are modified, and functions are re-grouped in different namespaces or files. Therefore, the restructuring is just like a new release of software and the same method can be applied as discussed in Section 2.8. For example, in the "Sed" program (of the UNIX utilities) release 3.02 to release 4.0.6 took 5 years, almost every major function changed, new developers work on the project and changes were significant (Do et al.,2005). Our results on the Sed program are shown in Section 2.6.

2.12 Conclusions and Future Work

Discovering the origin of a fault (Brodie et al., 2005; Lee and Iyer, 2000) is an arduous task; it soaks up 30% to 40% of the time required to do corrective maintenance (Proprietary Workshop, 2008) -- despite the fact that 10%-20% of a program's code is deemed responsible for 80% of the faults (Gittens et al., 2005; Ostrand et al., 2005) and 50%-90% of field failures are rediscoveries of previous faults (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003). A number of techniques proposed for deployed software focus on: the classification of field profiles into failed or successful executions (Bowring et al., 2004; Haran et al., 2007); clustering field profiles (Podgurski et al., 2003); rediscovery of crashing failures (Brodie et al., 2005; Lee and Iyer, 2000); and statistical debugging (Chilimbi et al., 2009).

This paper describes a new technique (called F007) for identifying faulty functions from a given "failed" execution trace. F007 trains the decision tree algorithm C4.5 on function-calls extracted from a collection of failed traces to identify faulty functions in a new failed trace (see Section 3). F007 deals with both crashing and non-crashing faults. This technique is beneficial for deployed systems because most of the failures are rediscoveries of the same or similar faults originating from the same area of the source code (see Section 2.1). F007 is also useful when only a few traces are available and a collection of many traces (e.g., pass-fail) is not feasible. In contrast, closely related techniques find the fault in a trace at a coarser level (Podgurski et al., 2003) such as files, discovering only those rediscovered faults causing crashes (Brodie et al., 2005; Lee and

Iyer, 2000) , or require both passing traces and failing traces for a fault (Chilimbi et al., 2009).

We evaluated F007 on the Siemens suite (Hutchins et al., 1994), the Space program, and the four UNIX utilities (i.e., Flex, Grep, Gzip and Sed) (see Section 2.6). On training F007 on 1-25% of the failed traces, F007 can identify faulty functions in: (a) 75% to 90% of the failed traces in the test set of the UNIX utilities by reviewing 5% or less of the code (see Section 2.6.2); (b) 70% of the failed traces in the test set of the Siemens suite by reviewing 20% or less of the code (see Section 2.6.2); and (c) 96% of the failed traces in the Space program by reviewing 5% or less of the code (see Section 2.6.2). There is no direct quantitative comparison of F007 with the related techniques (see Section 2.10); however, if compared with the closest techniques (Di Fatta et al., 2006; Jones and Harrold, 2005) (see Figure 20) identifying faulty functions using a collection of pass-fail traces, then F007 still identifies faulty functions with 20% better accuracy at a finer grain level of an individual trace. F007 makes this identification using previous failed traces of (at least one fault of) the same function and focuses on corrective maintenance; whereas other techniques require pass-fail traces and focus on testing.

During our experiments, we found that: (a) patterns (combinations) of function-calls (see Section 2.3.1) do not discover faults better than single function-calls when used with classification algorithms (e.g., decision tree; see Section 2.6.1); (b) the use of only the "function entry" or the "function exits" in the function-call level trace would suffice to discover faults with the same accuracy as that when both the "function entry and exit" are used together (see Section 2.6.4) – thus, reducing the overhead (size, time) of a trace to half; and (c) faults in the same function occur with similar sequence of function-calls because F007 can identify same faulty functions with new faults by using the knowledge of previous faults in the same function (see Section 2.6.2 and Section 2.7).

We also evaluated F007 on a large commercial software system of over 20 million LOC and 200K+ functions (see Section 2.7). F007 can identify faulty functions in the subsequent releases by using the failed traces of previous releases with: (a) approximately 10-60% failed traces on the review of 10% of the code (see Section 2.6.2) in the UNIX

utilities; and (b) approximately 30-60% failed traces on the review of 3% in the large commercial application (see Section 2.7.6). Similarly, if F007 is trained on 10-25% of the failed traces of a release, then F007 can identify faulty functions in approximately 70-80% of the remaining failed traces of the same release of the large program: this identification requires 1-3% of the code review (see Section 2.7.6). In the case of the large system, we empirically found that rarely occurring function-calls or function-calls with the similar frequencies among traces can be removed without affecting the accuracy (see Section 2.7.6); this is similar to what Yuan et al. (2006) found on another large system (Windows XP). This can further facilitate in reducing the size of a function-call trace. Moreover, no direct quantitative comparison of performance of F007 on the large program is possible with any other techniques because of its large size and approach of F007 in discovering faulty functions (or components).

Though F007 cannot identify the faults in the new faulty functions, this limitation is compensated by the fact that 50-90% of the field failures are rediscoveries of the previous faults (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) and originate from the same 20% area of code (Gittens et al., 2005; Ostrand et al., 2005). For future work, we are planning studies to overcome this limitation by employing mutants to generate faulty functions for data preparation. This would allow F007 to identify all the faulty functions if the faults in the same function occur with similar sequence of function-calls.

Thus, we conclude that using function-call traces of at least one fault in the same function from previous releases or the same release, we can identify faulty functions in majority of the traces of new failures. This has been verified on many subject programs of this study.

2.13 References

Agrawal, H.; Horgan, J.R.; & London, S.; Wong, W.E.; “Fault Localization using Execution Slices and Dataflow Tests”. *Proc. Int’l Softw. Symp. on Reliability Eng.*, France, Oct., 1995, pp.143-151.

Bowring J.F.; Rehg J.M.; and Harrold. M.J; “Active Learning for Automatic Classification of Software Behavior”, *SIGSOFT Soft Eng. Notes* Vol. 29, No. 4, ACM, US, Jul. 2004, pp. 195-204.

Brodie, M.; Sheng Ma; Lohman, G.; Mignet, L.; Modani, N.; Wilding, M.; Champlin, J.; and Sohn, P.; “Quickly Finding Known Software Problems via Automated

Symptom Matching". *Proc. 2nd Int'l Conf. on Autonomic Computing, Seattle, USA*, June 2005, pp. 101-110.

Chapin. N, "Do We Know What Preventive Maintenance Is?, *Proc. of Int'l Conf. on Soft. Maintenance*, IEEE CS, Washington , USA, October, 2000, pp. 15-17.

Chen M.; Accardi A.; Kiciman E.; and Fox A.; Patterson D.; and Brewer E.; "Path-based Failure and Evolution Management". *Proc. Int'l Symp. on Networked Systems Design and Implementation*, San Francisco, USA, March 2004, pp. 309-322.

Chilimbi, T. M.; Liblit, B.; Mehra, K.; Nori, A. V.; and Vaswani, K; "HOLMES: Effective Statistical Debugging via Efficient Path Profiling". *Proc. 31st Intl. Conf. on Softw. Eng*., IEEE CS, Canada, May, 2009, pp. 34-44.

Dallmeier, V.; Lindig, C.; and Zeller, A.; "Lightweight Defect Localization for Java". Proc. 19th European Conf. on Object-Oriented Programming, Springer LNCS, Glasgow, UK, Aug. 2005, pp. 528-550.

Di Fatta, G.; Leue, S.; and Stegantova, E; "Discriminative Pattern Mining in Software Fault Detection". *Proc.3rd Int'l Workshop on Softw. Quality Assurance*, ACM, Oregon, USA, Nov. 2006, pp. 62-69.

Ding, X.; Huang, H.; Ruan, Y.; Shaikh, A.; and Zhang, X.; "Automatic Software Fault Diagnosis by Exploiting Application Signatures". *Proc. 22nd Conf. on Large Installation System Admin., San Diego,* CA, USA, Nov., 2008, pp. 23-39.

Do, H.; Elbaum, S. G.; and Rothermel, G.; "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact". *Empirical Softw. Eng.*, Vol. 10, Springer, Oct. 2005, pp. 405-435.

Elbaum, S., Kanduri, S., and Andrews, A. "Trace Anomalies as Precursors of Field Failures: An Empirical Study". *Empirical Softw. Eng.*, Vol.12, No.5, Springer, Oct. 2007, pp. 447-469.

Etrace (Runtime Tracing Tool): http://ndevilla.free.fr/etrace/; March, 2008

Gittens M.; Kim Y.; and Godwin D.; "The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software". *Proc. 29th Int'l Computer Softw. and Appl. Conf.*, IEEE CS, Edinburgh, Scotland, July 2005, pp. 179-185.

Haran, M.; Karr, A.; Last, M.; Orso, A.; Porter, A.A.; Sanil, A.; Fouche, S.; "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks". *IEEE Trans. on Softw. Eng.* Vol. 33, No.5, May, 2007, pp. 287-304.

Hutchins, M.; Foster, H.; Goradia, T.; Ostrand, T., "Experiments on The Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria". *Proc. 16th Int'l Conf. on Softw. Eng.*, IEEE, Sorrento, Italy, May,1994 ,pp.191-200.

Jones, J. A. and Harrold, M. J., "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique". *Proc. 20th Int'l Conf. on Automated Softw. Eng.*, IEEE/ACM, CA, USA, 2005, pp.273-282.

Lee M. G. and Jefferson T. L.; "An Empirical Study of Software Maintenance of a Web-based Java Application". *Proc. Int'l Conf. on Soft. Maintenance*, IEEE, Budapest, Hungary, Sep., 2005, pp. 571-576.

Lee, I.; Iyer, R., "Diagnosing Rediscovered Problems Using Symptoms". *IEEE Trans. on Sofw. Eng.*, Vol. 26, No. 2, Feb, 2000, pp.113-127.

Liu, C. and Han, J., "Failure Proximity: A Fault Localization-based Approach". *Proc. of the 14th SIGSOFT Symp. on Foundations of Softw. Eng.*, ACM, Portland, USA, Nov. 2006, pp. 45-56.

Liu, C.; Yan, X.; Fei, L.; Han, J.; Midkiff, S. P.; "SOBER: Statistical Model-Based Bug Localization". *SIGSOFT Softw. Eng. Notes*, Vol 30, No.5, ACM, USA, Sep., 2005, pp. 286-295.

Mannila, H.; Toivonen, H; Inkeri,V.; "Discovery of Frequent Episodes in Event Sequences". *Data Mining and Knowledge Discovery*, Vol. 1, No. 3, Springer, Jan 1997, pp. 259-289.

Marques de Sá, J., P.; *Applied Statistics Using SPSS, STATISTICA, MATLAB and R*, 1st ed., Springer, Aug., 2003.

Murtaza, S.,S.; Gittens, M.; Li, Z.; Madhavji, N.,H.; "F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field". *Proc. Conf. of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ACM, Canada, Nov. 2010, pp. 61-75.

Ostrand T. J., Weyuker E., and Bell R. M., "Predicting the Location and Number of Faults in Large Software Systems". *IEEE Trans. on Softw. Eng.*, Vol. 31, No. 4, 2005, pp. 340-355.

Podgurski, A.; Leon, D.; Francis, P.; Masri,W.; Minch, M.; & Sun, J.; Wang, B, "Automated Support for Classifying Software Failure Reports". *Proc. Intl. Conf. on Softw. Eng.*, IEEE CS, Portland, US, May, 2003, pp. 465-475.

Polat, K. and Güneş, S; "A Novel Hybrid Intelligent Method Based on C4.5 Decision Tree Classifier and One-Against-All Approach for Multi-class Classification Problems". *J. of Expert Syst. Appl.,* Vol. 36, No.2, Pergamon Press, Mar.2009, pp.1587-1592.

Proprietary workshop on large commercial software, Sep., 2008.

Refaat, M.; Data Preparation for Data Mining using SAS, Elsevier, 2007.

Quinlan, J. R.; *C4.5: Programs for Machine Learning,* Morgan Kaufmann Publishers, 1993.

Refaat M., Data Preparation for Data Mining using SAS, Elsevier, 2007.

Schach S. R.; Jin B.; Yu L.; Heller G. Z.; and Offutt J.; "Determining the Distribution of Maintenance Categories: Survey versus Measurement". Empirical Soft. Eng. Vol. 8, No. 4, Springer, Dec., 2003, pp. 351-365.

Siemens Suite: http://www-static.cc.gatech.edu/ aristotle/Tools/ subjects/, March, 2008.

Sousa M. J., "A Survey on the Software Maintenance Process", Proc. of 14th Int'l Conf. on Soft. Maintenance, IEEE CS, Washington, March, 1998, pp. 265-274.

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

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; and Wesslén, A;. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, USA, 2000..

Wong, W. E. and Qi, Y.; "Effective Program Debugging Based On Execution Slices and Inter-Block Data Dependency". *J. Syst. Softw.,* Elsevier, Vol.79, No. 7, July, 2006, pp. 891-903.

Wong, W.E.; Yu Qi; Lei Zhao; Kai-Yuan Cai, "Effective Fault Localization using Code Coverage". *Proc. 31st Int'l Conf. on Comp. Softw. & App.*, IEEE, China, July, 2007, pp.449-456.

Wood A., "Software Reliability from the Customer View". *Computer*, Vol. 36, No. 8, IEEE CS, Aug., 2003, pp.37-42.

Yuan, C.; Lao, N.; Wen, J.; Li, J.; Zhang, Z.; Wang, Y.; and Ma, W; "Automated Known Problem Diagnosis with Event Traces". *SIGOPS, OS. Syst. Rev.*, Vol. 40, No. 4, ACM, USA, Oct., Aug. 2006, pp. 375-388.

Zhang, Z.; Jiang B., and Wang, X., "Capturing Propagation of Infected Program States". Proc. *Intl. Conf.* on *Foundations of Soft Engg.*, ACM, Netherlands, 2009, pp. 43-52

Zheng A.X.; Jordan M.I., Liblit, B.; and Aiken, A; "Statistical Debugging of Sampled Programs". *Advances in Neural Info. Processing Syst.*, MIT Press, Cambridge, MA, US, 2004, pp. 9-18.

Chapter 3

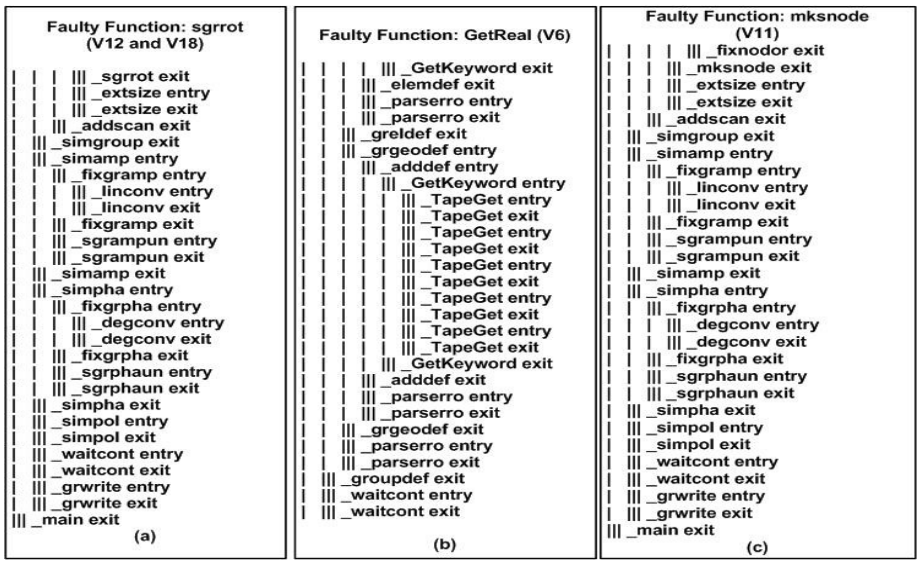
3 Using Mutants to Discover New and Rediscovered Field Faults by Exploiting the Similarity of Traces among Different Faulty Functions

3.1 Introduction

Scientific literature indicates that corrective maintenance of software consumes 30-60% (Lee and Jefferson, 2005; Schach et al., 2003) of the time of maintenance activities. Typically, maintainers collect data (such as execution traces) related to software failures in order to fix the faults. For cost reduction and quality purposes, organizations of such applications as Mozilla, NetBeans, Microsoft Visual Studio.NET and others have employed automated means to collect and report failure-data.

While such automation makes data collection and reporting practical from numerous sources, it can also overwhelm developers because manually interpreting such reports and identifying fault origin is resource draining for large systems with huge user bases (Podgurski et al., 2003). Data shows that it can soak up 30%-40% of corrective maintenance time (Proprietary Workshop, 2008).

Thus, various researchers have focused on reducing the time spent in the discovery of faults. Prior research includes: (a) classification of field traces as failed or successful (e.g., using decision trees (Haran et al., 2007), using Markov models (Bowring et al., 2004); (b) classification of rediscovered crashing failures (e.g., by matching symptoms with previous faults (Brodie et al., 2005; Lee and Iyer, 2000)); (c) clustering traces relating to coarse grain code such as files (Podgurski et al., 2003); (d) using statistical debugging to identify a fault from passing traces and failing traces of that fault (Chilimbi et al., 2009; Liu and Han, 2006); and (e) our earlier work (called F007) (Murtaza, et al., 2010) that identified rediscovered faulty functions from failed traces. Prior techniques either (i) require a collection of passing and failing traces, which is resource-intensive for deployed software, or (ii) they focus only on rediscovered faults, which means new faults cannot be discovered.



The 'entry' and 'exit' points represent function entry and exit, respectively.

Figure 23: Failed function-call level execution traces for faults in function "sgrrot", "GetReal" and "mksnode" of the Space program.

Different functions in a computer program have different purposes, but they are programmed to cooperate in order to achieve specific goals according to stakeholder scenarios. It is possible that though two goals (say A and B) are distinct, there are partially overlapping execution paths in achieving these two goals. That is, some functions and the sequence in which they are executed are similar in attempting to reach the distinct goals A and B. Also, two distinct goals X and Y can have completely non-overlapping execution paths; that is, functions executed in achieving goals X and Y are different. Thus, the program traces resultant from the execution of similar function sequences (on the paths to achieve goals A and B) will be similar, and program traces due to goals X and Y will have different function sequences.

In Figure 23, we show the ends of traces of faults in functions of the Space program (Do et al., 2005) – an interpreter for an antenna array definition language written for the European Space Agency. Figure 23 shows that when two different faults (V12 and V18) occur in the function "sgrrot" then program traces of failing test cases are exactly the same; and when a fault (V11) occurs in the function "mksnode" (part 'c') then the trailing

end of the failing trace is the same as the failing trace of faults in function “sgrrot” (part ‘a’). On the contrary, Figure 23 also shows that the program traces of a fault (V6) in the function “GetReal” (part ‘b’) are completely different from the traces of faults in the functions “sgrrot” and “mksnode”. These observations show that (failing) traces of faults in some functions (e.g., “sgrrot” and mksnode”) are similar to each other, and traces of faults in some functions are completely different from one another.

These initial observations warrant further empirical investigation because—if there is a similarity in the faulty traces of related functions then—using the traces of artificial faults (e.g., manually or automatically seeded faults) of functions, we can identify the same faulty functions in the traces from actual program executions by users. In practice, artificial faults can be seeded into functions during (or after) software testing before deploying a software application, and traces of artificially faulty functions can be collected by running test cases. Literature suggests that we can identify the majority of the real faulty functions using artificially faulty functions because: (a) 50-90% of the faults are rediscoveries of previous faults (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) —implying that function-call patterns will be similar for the same faulty functions; and (b) 20% of the code is responsible for 80-100% of the faults (Gittens et al., 2005; Ostrand et al., 2005)—implying that function-call patterns will be similar for the faults originating from the same area of code (functions). If the majority of faulty functions in field traces can be quickly identified then the time spent in corrective maintenance would be reduced.

In our previous work (called F007) (Murtaza, et al., 2010), we showed empirically that different faults in the same function occur with similar function-calls. We observed that by using the function-call level traces of at least one fault of a function, we can discover the same faulty function from the traces of different faults in that function (Murtaza, et al., 2010). This characteristic was observed on the function-calls of only previously known faulty functions of the subject programs; which represented a small percentage of total functions of the subject programs. In this paper, considering the observations of Figure 23 (that traces of different faulty functions are similar), we extend the scope of our

previous empirical investigation to all the functions (i.e., making them artificially faulty) by addressing the question:

(Q1) Are the function-call level traces of some faulty functions similar and that of some other faulty functions different?

Furthermore, F007 (Murtaza, et al., 2010) requires a historical collection of failed traces of the actual faulty functions in order to discover those functions in the traces of (crashing and non-crashing) field failures. Though useful for identifying faulty functions in the field traces, this also limits the use of F007 to only the set of previously known faulty functions. This paper overcomes the limitation of F007 by proposing a new strategy to identify both new and old faulty functions in the failed traces. This new strategy, called *F007-plus*, uses (a) failed traces of mutants (Offutt et al., 2001) (i.e., automatically seeded faults generated by modifying statements) – this was not the case in the previous F007 approach (which, from hereon, is called *F007-basic* to avoid confusion)—and (b) failed traces of previously known faulty functions in order to identify the new and old faulty functions in traces. The use of mutants in F007-plus, in fact, facilitates F007-basic (Murtaza, et al., 2010) to discover new faulty functions. Thus, the second research question that follows from this mutant based strategy is:

(Q2) Can the faults generated using mutants be used to discover the actual faults?

These two questions posed in this paper are new and have not been dealt with in the literature before. As mentioned earlier, these questions are important because we can reduce the time and effort spent in corrective maintenance by using artificial faults (i.e., mutants) to discover the actual faults. This paper investigates the answers to these questions by using Space (Do et al., 2005), Flex (Do et al., 2005), Grep (Do et al., 2005), Gzip (Do et al., 2005) and Sed programs (Do et al., 2005) as the subject programs of this study. These programs were selected because of their commercial or production- level usage, reasonable sizes, and use by other fault discovery and testing related techniques (Andrews et al., 2005; Jones and Harrold, 2005; Wong et al., 2007; Zhang et al., 2009).

The contributions of this paper are:

(a) The finding that traces (function-calls) of different faults in a group of related functions are similar; and traces of faults in one group of functions are different from traces of faults in another group of functions. This answers the first research question (Q1).

(b) When using F007-plus, faulty functions in approximately 30-80% of the failed traces of the latest release of the subject programs can be identified by reviewing 20% of the code: in this identification F007-plus used traces of mutants of the latest releases and the failed traces of previous releases. This answers the research question (Q2). If F007-plus is compared against F007-basic (which only uses traces of previous releases) then F007-plus improves upon F007-basic by 10-60% by reviewing of 20% of the code.

(c) If compared with third party techniques in the literature then F007-plus: (i) does not require the knowledge of passing and failing traces (Chilimbi et al., 2009; Liu and Han, 2006) and (ii) can identify finer–grain faulty functions from both crashing and non-crashing failures (Brodie et al., 2005; Lee and Iyer, 2000). Also, this paper distinguishes itself by investigating: (i) the use of mutants in discovering actual faults; and (ii) the similarity of traces among different faulty functions. The collective contributions of this paper add to the advancement of scientific knowledge, and were not discussed in the literature before.

The rest of the paper continues as follows: in Section 3.2, we describe related work; Section 3.3 describes the differences between F007-basic and F007-plus; Section 3.4 explains the fundamentals of mutation and decision tree as used in F007 and F007-plus; Section 3.5 describes the steps of F007-plus strategy; Section 3.6 describes the execution time; Section 3.7 finds answers to the issue of similarity of function-calls (Q1) and the effect of making every function artificially faulty using mutants (Q2); Section 3.8 evaluates the strategy F007-plus (Q2) and improves the answer to (Q2); Section 3.9 summarizes all the results of this paper; Section 3.10 compares F007-plus against other techniques; Section 3.11 explains threats to validity of the investigated questions; and Section 3.12 explains the conclusion and future work arising from this study.

3.2 Related Work

This section describes closely related techniques such as: fault discovery techniques for in-house faults (e.g., evaluating statement coverage to discover faulty statements (Jones and Harrold, 2005; Wong and Qi, 2006, Wong et al., 2007); fault discovery techniques for field failures (e.g., rediscovering known crashing problems (Lee and Iyer, 2000; Liu and Han, 2006) and the use of mutation (e.g., for measuring test case efficacy (Mayer and Schneckenburger, 2006; Do and Rothermel, 2006).

3.2.1 Fault Discovery Techniques for Inhouse Faults

Many researchers have proposed techniques for discovering fault locations by using the difference between passing traces and failing traces pertaining to a fault such as: discovering faulty statements using statement level traces (e.g., Agrawal et al. (1995), Wong and Qi (2006), Jones and Harrold (2005), Zhang et al. (2009) and Wong et al. (2007)); discovering faulty functions using function-call traces (e.g., Di Fatta et al. (2006)); discovering faulty classes using function-call traces (e.g., Dallmeier et al. (2005)); and identifying assertions (e.g., null pointer checks) using statistical debugging (e.g., Zheng et al. (2004) and (SOBER) Liu et al.(2005)).

These techniques are suitable for in-house testing but not for deployed software because: (a) different customer usages can cause many different normal execution paths that are not observed in the passing executions of in-house testing, and it is not feasible to collect many passing traces from customers due to overhead involved in the trace collection;(b) they require a collection of failing traces related to the same fault, but failure traces in the field do not necessarily materialize for the same fault; (c) in deployed software, often only a few traces (at the time of fault) are available due to the overhead incurred in the trace collection, and sometimes it is not known if a trace is passing or failing; and (d) the statistical debugging-based techniques require the knowledge of a type of a fault for instrumentation, and if a fault is not found, another type of assertion (e.g., null pointer check) is instrumented in source code and the process continues till the fault is found—not feasible to disrupt the execution of deployed software for so long.

3.2.2 Fault Discovery Techniques for Field Failures

Podgurski et al. (2003) form clusters of execution traces of the field failures based on common faulty source files. The granularity in the Podgurski et al. approach is a faulty file where the majority of the clusters encompasses failed traces with different files, making it not suitable for the discovery of finer-grained (e.g., function) origin of fault. In contrast, in this paper we focus on discovering faulty functions from only a field trace using previous failing traces and traces of mutants.

Liu and Han (2006) cluster failing runs according to the rank list of assertions obtained using the statistical debugging tool SOBER (Liu et al., 2005). Liu and Han (2006), require a collection of passing traces and failing traces for the same fault to discover its origin. Their work also suffers from the limitations of statistical debugging-based techniques and requirement of pass-fail traces (see Section 3.2.1). In this paper, we focus on discovering faulty functions by using only failed traces by avoiding such limitations.

Another statistical debugging tool, HOLMES (Chilimbi et al., 2009), identifies suspicious fault locations from the traces containing executed paths of deployed software. This tool can only be applied to server side applications, because they (Chilimbi et al., 2009) have to redeploy software components with instrumentation of selected functions to collect passing traces and failing traces pertaining to one fault. The use of passing-failing traces is also a limitation, as discussed in Section 3.2.1. Also, in some cases, this may not be feasible for servers too due to the redeployment of instrumented software components on running systems.

Elbaum et al. (2007) propose a technique that compares the field failure traces (function sequences) to in-house passing traces in order to anticipate the occurrence of a failure such that data collection for a defect in the field can be started. Bowring et al. (2004) and Haran et al. (2007) develop a technique based on the Markov model (Bowring et al., 2004) and the decision tree (Haran et al., 2007) to characterize (statement or branch level) executions as being passing or failing runs. These techniques complement our work in discovering faulty functions in the following ways: (a) if traces collected from the field also contain some of the passing traces, they can be filtered by using the techniques

proposed by Haran et al. (2007) and Bowring et al. (2004); or (b) failed traces from the field can be collected directly from the field using Elbaum's et al. (2007) technique. Subsequently, F007-pluse can use the filtered (failed) traces to localize faulty functions.

Brodie et al. (2005) use string matching to group one function-call trace of a crash with other groups of function-call traces for different known crashes. Lee and Iyer (2000) propose a technique to classify a rediscovered crashing failure by literal matching of its function-call trace with already known failure traces. They consider a variety of heuristics to match several function-call paths followed by the same fault. In this paper, we address the more difficult problem of non-crashing failure classification, where a user may notice a failure well after the execution of faulty code. We focus on discovering faulty functions in both the non-crashing (e.g., logical error) and crashing failure (e.g., segmentation fault) traces at the fine-grained level of faulty function.

Yuan et al. (2006) employ support vector machines (a classification algorithm in machine learning) to determine the root causes of a problem (e.g., unable to browse on Internet explorer) confronting a user of a system on the basis of execution traces of software. Chen et al. (2004) describe a technique based on the decision tree and the association rule to diagnose faulty components (e.g., a web server) in large distributed systems. Ding et al. (2008) also propose a technique to identify faults occurring due to configuration of software (e.g., large log files) by contrasting failing and passing runs. These techniques also complement our work; for example, these techniques can identify fault at the system level (e.g., memory overload), after which F007-plus can identify logical faults (e.g., infinite loop).

3.2.3 Fault Discovery Using Mutation

Mutants are automatically generated variants (faulty version) of a program obtained by applying mutation operators to the source code (Offutt et al., 2001): this is called mutation (Andrews et al., 2005; Offutt et al., 2001). For example, mutation operators include changing an arithmetic operator with another in a statement, negating a decision in if or while statements, or deleting a statement. Moreover, mutation analysis is a

measure of quality of test cases (Offutt et al., 2001). Mutation and related concepts are further explained in Section 3.4.2.

Mutation analysis has mostly been used to measure, enhance and compare the effectiveness of testing strategies. For example, Mayer and Schneckenburger (2006) use mutation analysis to compare the effectiveness of all the adaptive random testing techniques in detecting failures. Similarly, Do and Rothermel (2006) employ mutation analysis to evaluate the ability of several test case prioritization techniques in improving the fault detection rate on Java programs. Test case prioritization is used to reduce the cost of regression testing by running important test cases first—i.e., test cases which have more chances of detecting faults (Do and Rothermel, 2006). Andrews et al. (2006) use mutation analysis to compare the cost-effectiveness of data and control flow coverage criteria (i.e., Block, Decision, C-Use, and P-Use). Andrews et al. have also empirically determined that mutation faults are similar to real faults (Andrews et al., 2005; Andrews et al., 2006), but different from hand seeded-faults (Andrews et al., 2005). Hao et al. (2005) use mutants as faulty versions of a program to evaluate fault localization techniques. In contrast, the focus of this paper is on using the faults generated from mutation to discover the origin of real faults, and on using mutants to determine the similarity of function-calls of different faults.

3.2.4 Research Gap

Comparable studies in the literature focusing on field failures find faults in traces at coarse-grain levels (e.g., file) (Podgurski et al., 2003), discover only crashing faults (Brodie et al., 2005; Lee and Iyer, 2000) require a collection of passing and failing traces for a fault (Chilimbi et al, 2009; Liu and Han, 2006) and F007-basic (Murtaza, et al., 2010) (overviewed in the next section) identifies only *rediscovered* faulty functions from only a failed field trace. Based on this, we describe the research gap as: (i) identifying *new* faulty functions from only a failed trace of crashing and non-crashing failures and (ii) discovering actual faults using mutants, which helps in preparation of failed traces without affecting deployed systems.

This paper distinguishes from previous work as in: (a) it proposes a new strategy (F007-plus) that improves F007-basic in discovering new and rediscovered faulty functions from failed field traces; (b) it investigates whether mutants can be used to discover actual faults; and (c) it investigates whether function-call traces of different faulty functions are similar.

These distinguishing factors are important because: (a) F007-plus can discover faulty functions when only a few traces of failures are available from deployed systems; and (b) we can use the function-call level traces of mutants of approximately 20% functions to identify faulty functions in approximately 80% of the actual failed traces, if the function-call traces of different faulty functions are similar.

3.3 F007-basic and F007-plus Overview

In this section, we explain the fundamental differences of F007-basic and F007-plus. First we describe the steps of F007-basic, showing how F007 works, in Section 3.3.1. Second, we identify the steps of F007-plus in Section 3.3.2.

3.3.1 F007-basic

F007-basic identifies rediscovered faulty functions (Murtaza, et al., 2010) from the function-call level traces of field failures. The steps of F007-basic are explained below:

Step 1: F007-basic first extracts function-calls and their occurrences from the given function-call level traces [15] of prior failures of a program. F007 then labels function-calls and their occurrences of each trace with its corresponding faulty functions.

Step 2: Secondly, F007-basic trains the decision trees using one-against-all (Witten and Frank, 2005) approach on the extracted function-calls and their corresponding faulty functions, identified in Step1. In the one-against-all approach, one decision tree is trained for each faulty function.

---

[15] Function-call level trace is shown in Figure 23 where the “function entry” shows when control enters the function and the “function exit” shows when the control leaves the function.

Step 3: Thirdly, when a new failed trace arrives, F007-basic also extract function-calls and their occurrences from that trace and provides it to the trained decision trees. Each decision tree predicts a faulty function with a probability. Functions are then arranged in decreasing order of the probability with the intuition that the top most functions are more likely to be faulty than the lower ones in the list.

3.3.2 F007-plus

The F007-plus strategy facilitates F007-basic to discover new and rediscovered faulty functions. A scenario for fault discovery using F007-plus in a failed trace of a deployed system is depicted in Figure 24. Figure 24 shows that:

(1) When a fault occurs in the deployed system, a function-call level trace is captured. The captured trace is then passed on to a "fault locator" as shown in Figure 24.

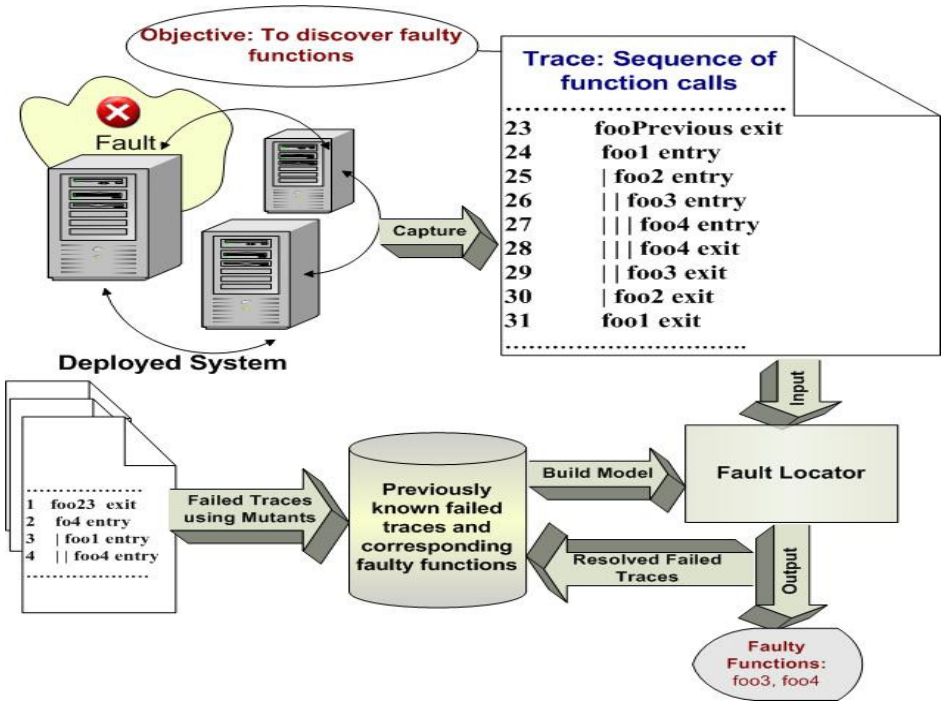


Figure 24: A scenario of fault discovery in failed traces of deployed software.

(2)The "fault locator" identifies faulty functions in the captured failed trace of the deployed software system by using the decision tree (Witten and Frank, 2005) built on a collection of failed traces of mutants (artificial faults) and prior faults. This strategy

of using mutant and prior faults is called F007-plus. F007-plus works as follows: (a) F007-plus first identifies approximately 20% (or more) of the suspected faulty functions in a software release, and generates mutants for those suspected functions; and (b) F007-plus then collect traces on mutants of the suspected functions, and execute F007-basic on traces of mutants and prior releases to discover new an old faulty functions. The use of mutants facilitates F007-basic to discover majority of the new faulty functions. Thus, the fault locator, shown in Figure 24, is based on the improved strategy (F007-plus) of mutants and prior faults to train the decision trees.

(3) Once faulty functions in the captured failed trace are correctly identified, the trace is added to the collection of the failed traces. The fault locator will then be able to resolve potential failed traces based on the updated knowledge of the failed traces.

Thus, F007-plus strengthens F007-basic further in identifying faulty functions. The use of mutants in F007-plus also facilitates in investigating questions posed in Section 3.1.

3.4 Fundamentals

This section provides the background necessary to understand the rest of the paper. In Section 3.4.1 we first characterise the subject programs used in the study. Second, in Section 3.4.2, we describe the fundamentals of mutation and draw examples of mutants from the subject program. Third, in Section 3.4.3, we explain how decision trees are generated using one-against-all approach, and show examples on mutant traces of the subject programs.

3.4.1 Subject Programs

We used the Space program (Do et al., 2005) and four open source UNIX utilities (Do et al., 2005) (i.e., Flex, Grep, Gzip and Sed) for our experiments. Space is a C program, an interpreter for an antenna array definition language written for the European Space Agency, and the faults were found during actual development. The UNIX utilities are well known commercial applications and the faults in the UNIX utilities were hand seeded (Do et al., 2005) but a specific procedure was followed to keep them realistic (Do et al., 2005). For example, the faults were inserted at the changes between the source

codes of different releases. Space program and the UNIX utilities are made available by Do et al. (2005) at the subject infrastructure repository (SIR). Space program has been used in a number of major fault localization studies; e.g., classification of field traces as failed or successful (Bowring et al., 2004), and finding faulty statements in a program during in-house testing (Jones and Harrold, 2005; Wong and Qi, 2006). UNIX utilities were also used in different studies; e.g., identifying faulty statements using edge profiles (Zhang et al., 2009). Table 11 characterizes these programs in detail.

Table 11: Characteristics of the subject programs.

Flex, Sed, Grep and Gzip are well known UNIX utilities. Space is an interpreter for an antenna array definition language written for the European Space Agency.						
Releases for Flex: R1=2.4.7, R2= 2.5.1, R3=2.5.2, R4=2.5.3, R5=2.5.4. Releases For Grep: R1=2.2, R2= 2.3, R3=2.4, R4=2.4.1. Releases for Gzip: R1=1.1.2, R2= 1.2.2, R3=1.2.4, R4=1.3. Releases for Sed: R1=2.0.5, R2= 3.01, R3=4.0.6, R4=4.0.7, R5=4.1.5. Space and seven programs in the Siemens suite have only one release.						
Program	Test Cases	LOC (excludes comments & blank lines)	Functions	Faulty Functions	Faulty Versions	Failed Traces
Flex	567	8250-9831	151-169	3-12 (26)	4-16 (45)	7-362 (877)
Grep	809	8484-9041	142-150	2-4(9)	3-5(15)	11-247 (659)
Gzip	214	4032-5103	89-111	3-6(13)	3-6(16)	14-50 (99)
Sed	370	4711-9226	115-183	1-4 (10)	3-5 (18)	60-141 (465)
Space	13585	5767	136	26	34	71958

UNIX utilities come with different releases and have several faults in each release; whereas the Space program has one release and many different faults. There are twenty programs in Table 11, if we consider each release as one program. Each program consists of one original version -- deemed correct because it passes all the test cases--and several faulty versions -- deemed fail because they fail on one or more of the given test cases. A faulty version is a variant of the original version by one fault—that is, one fault is equivalent to one faulty version. A fault is equivalent to incorrect statements in the code.

In Table 11, the second row shows the release-numbers for each of the releases of the UNIX utilities used in our study. We have labeled each release from R1 to R5, which will be used in the following sections. In Table 11, first column shows the name of a program and the second column shows the number of test cases. In the UNIX utilities, the test cases were shared across releases. Third and fourth column in Table 11 show the lines of

code and the number of functions in a program. For the UNIX utilities third and fourth column represent minimum and maximum LOC or functions for the different releases of every program, respectively. For example, five releases of "Flex" have 8250 to 9831 lines of code and 151 to 169 functions. Similarly, the last three columns of Table 11 show the **minimum-maximum** number of distinct faulty functions, **minimum-maximum** faulty versions and **minimum-maximum** failed test cases for the releases of every program of the UNIX utilities. A number in the bracket of the last three columns for the UNIX utilities show the total number of **distinct** faulty functions, total number of faulty versions and total number test cases across all the releases of a program. For example, in the case of Flex program, Table 11 shows that there are 3-12 faulty functions from release 1-5 of the Flex program, and there are a total of 26 distinct faulty functions from release 1-5. Similarly, there are 4-16 faulty versions (a total of 25), and 7-362 failed traces (a total of 877) in all the five releases of the Flex program. The distinct faulty functions point out that there were cases when the same function was found faulty across different versions; that is, there were different faults in the same functions of different releases. We used Etrace[16] (2008) to collect the function-call level failed traces as shown in Figure 23. A failed trace was collected when a test case failed on the faulty version. A test case was considered failed when output of the same test case on the faulty version differed from the original version of the program. Following the documentation provided by Do et al. (2005) for the UNIX utilities and following the previous experiments (Zhang et al., 2009) on the UNIX utilities, we excluded those faulty versions (faults) which failed on more than 20% of the test cases. Thus, in Table 11, the faulty versions column excludes those versions for which traces could not be captured due to non-failure of a test case or due to the exclusion condition of more than 20% cases.

Note that, for the "Grep" program no test cases failed for all the faults (faulty versions) of release 2.4.2; that is., the fifth release for the "Grep" program provided by Do et al. (2005) was excluded. Similarly, the "Sed" program had seven releases but no test cases

[16] Etrace has a bug which prevents it from capturing traces of the segmentation faults. We fixed it to collect such traces.

failed for the release 1.18 and release 3.02; and. for “Gzip” no test cases failed on release 1.2.3. Accordingly, we have excluded these releases. Finally, for the Space program, version 1, 2, 32 and 34 had no failing test cases.

Finally, the difference between mutants and the actual faults of the Space program and the UNIX utilities is that mutants are automatically generated faults and they are not the faults found in development or seeded by human experts; mutants are described below.

3.4.2 Mutation

The term mutation refers to the generation of mutants (faulty variant) of a program by applying mutant operators (e.g., replacing an arithmetic operator with another operator in a statement). Mutants are automatically generated (virtual) faulty versions; whereas, faulty versions of the Space program and the UNIX utilities in Section 2.2.1 are the actual faults.

A mutant is considered dead (or killed) if the output of a test case on the mutant differs from the output of that test case on the original program (Offutt et al., 2001). Mutants which are not killed by test cases are called live mutants. Live mutants actually show inadequacy and weakness of the test suite in exposing faults (Offutt et al., 2001). If a test suite misses some control flow paths of a program then it would be weak in detecting mutants (i.e., faults) on those paths of a program. Sometimes mutants become equivalent to the original program and they cannot be killed (Offutt et al., 2001)—i.e., they produce the same output as the original program. Identifying equivalent mutants is a tedious task and it is an undecidable problem (Andrews et al., 2005; Offutt et al., 2001) —not even automatic solutions can identify all equivalent mutants (Offutt et al., 2001).

In this paper, we used a program developed by Andrews et al. (2005) to generate mutants for the code written in the C language. In order to generate mutants for a source file, they (Andrews et al., 2005) apply “mutation operators” (when possible) sequentially to each line of code. This results into one mutant for every valid application of a mutation operator on each line of code. Andrews et al. (2005), based on the research on mutation operators, use the following four classes of mutation operators to generate mutants:

- First class replaces an integer constant C by 0, 1, -1, ((C)+1), or ((C)-1).
- Second class replaces an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class.
- Third class negates the decision in an `if` or `while` statement.
- Fourth class deletes a statement.

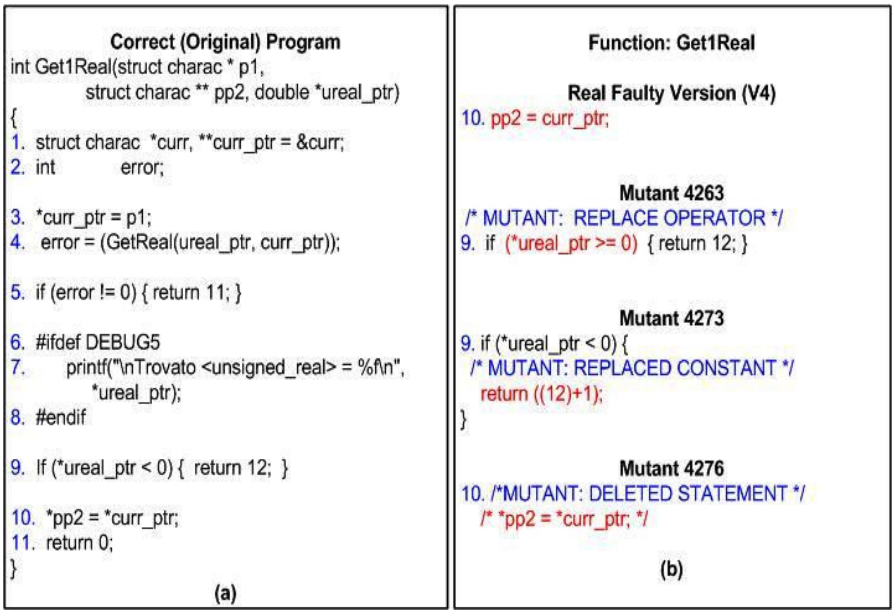


Figure 25: Correct source code of the function "Get1Real" of the Space program, its real faulty version and its faults generated using mutants.

In Figure 25, we show the examples of mutants for the Space program by using the above mutation operators. In Figure 25, part 'a' shows the source code of correct Space program for the function "Get1Real". Part 'b' of Figure 25 shows the faulty statements of the same function "Get1Real" found in real faulty version (version 4) of the Space program, and randomly selected mutants for the same function "Get1Real". These three mutants are obtained after applying three different mutant operators from the above list. Andrews et al. (2005) have also empirically determined the relationship of mutant faults with the actual faults. They found that mutant faults are close representative of the actual faults,

but they are different from hand seeded faults (when executing different techniques on mutants, handseeded and actual faults). They also observed that the hand seeded faults are harder to detect than the actual faults (2005).

The process of mutation can result in large number of mutants and it is computationally expensive to use all of them. For example, for the Space program (6218 LOC), the process of mutation can result in 12,262 mutants (i.e., 12,262 automatically generated faulty versions). It can be expensive to run test suites on all the mutants of a program and collect the failed traces for each mutant. For example, Dallmeier et al. (2005) found in their experiments on the SPEC JVM 98 Java programs suite (543 class files, total size 1.48 MB) that an instrumented program can take two orders of magnitude of a normal run (Dallmeier et al., 2005). This cannot be generalized; however, this shows an example of instrumented software run.

We therefore randomly selected only three mutants for every function of the subject programs. We used three mutants per function in order to avoid a situation of no failed traces for a faulty function, because sometimes a mutant does not compile or no test cases fail on mutants (Andrews et al., 2006). The use of three mutants does not guarantee the situation of no failed test cases, however generating more than three (e.g., five) would result in lots of mutants for a program. A mutant does not compile if a change in statement results in a compile time error; for example, deleting a variable declaration can result in compile time error. A test case does not fail on a mutant when the mutant is either a live mutant or an equivalent mutant.

Further, running all the test cases and collecting all the failed traces on mutants can be quite time and space consuming. Therefore, we decided to experiment by collecting a maximum of 10 failed traces per function, a maximum of 20 failed traces per function, and a maximum of 30 failed traces per function (i.e., 10 failed traces per mutant). The intuition is to determine the minimum number of failed traces necessary for predicting faulty functions while keeping it feasible for resource utilization. The function-call traces (see Figure 23) were again collected using Etrace (2008). We collected a failed trace if

the output of a test case on the mutant and the original version of a program (i.e., the Space or a release of the UNIX utilities) differed.

After collecting the failed traces of mutants, we trained the decision tree algorithm (Witten and Frank, 2005) on them to predict faulty functions in the failed traces of actual faults of the Space or the UNIX utilities. In data mining terminology, the failed traces of mutants form a training set and the failed traces of actual faults form a test set.

3.4.3 Decision Tree

We train the decision tree algorithm on a collection of failed traces. The intuition behind using the decision tree is that the decision tree algorithm can associate similar occurrences of function-calls to common faulty functions. The decision tree algorithm first requires the failed traces to be converted into a form on which the decision tree can be applied (Witten and Frank, 2005). This transformed representation of the failed traces is shown in Figure 26, part a.

	Functions							
	adddef	addscan	doubmax	elemdef	.....	versdef	waitcont	Faulty Functions
M373_T5605	4	0	0	1	.	0	1	adddef
M376_T2755	2	0	0	1	.	0	1	adddef
M025_T9506	1	0	0	1	.	0	1	Get1Real
M336_T13053	3	0	0	1	.	1	1	grgeodef
.....								.....
M228_T5123	4	1	1	1	.	0	1	simamp

(a) Original dataset for all categories

M373_T5605	4	0	0	1	.	0	1	others
M035_T9506	1	0	0	1	.	0	1	Get1Real
M336_T13053	2	1	0	1	.	0	1	others
M4276_T9507	1	0	0	1	.	0	1	Get1Real

(b) Dataset for "Get1Real" against all others.

Figure 26: Faulty functions and traces from mutants of the Space program.

Figure 26 shows selected examples of function-calls and failed traces of mutants for the Space program. A row represents a failed trace of a mutant and a cell represents the

occurrence of function-calls in that failed trace. The last column shows the known faulty functions for failed traces obtained from the corresponding mutant. For example, row one in Figure 26 shows that the function "adddef" occurred four times, "elemdef" occurred once, and "waitcont" appeared once in the failed trace of the test case 5605 (T5605) on the mutant 373(M373) of the Space program. The faulty function in this trace is the function "adddef". In data mining terminology, function-calls are independent variables and faulty function is a dependent variable.

The reason for selecting single function-calls as independent variables (as columns in Figure 26) lies in the empirical investigation of our earlier paper (Murtaza, et al., 2010), where we have empirically investigated that the patterns (sub-sequences) of function-calls do not yield better results than the single function-calls when used with the decision tree.

For example, consider an example of a pattern (sequence) of length three function-calls "adddef→elemdef→waitcont". This pattern is read as "adddef" precedes "elemdef" and "elemdef" precedes "waitcont" in the failed traces. If all such function-call patterns (Murtaza, et al., 2010) of different lengths are extracted from the failed traces and used with the decision tree to identify faulty functions, then the results are not better than the use of single function-calls with the decision tree (Murtaza, et al., 2010). Thus we consider the use of single function-calls and their frequencies with the decision tree in this paper, as shown in Figure 26. In short, the decision tree can build an accurate model of relationships of function-calls with single function-calls and their occurrences, and this model will be equal to the model of function-call relationships built using patterns (sequences) of different length of function-calls (Murtaza, et al., 2010).

Following the transformation of data, shown in Figure 26, we trained the decision tree algorithm on it using the one-against-all approach (Witten and Frank, 2005). In the one-against-all approach, a dataset (as in part 'a' of Figure 26) with M categories of

dependent variable[17] (faulty functions) is decomposed into M new datasets with binary categories. Each new binary dataset 'Di' has category 'Ci' (where i = 1 to M) labeled as positive and all other categories labeled as negative. An example of a dataset of a faulty function "Get1Real", against all "others" faulty functions, is shown in part b of Figure 26. It again shows a random selection of example traces (to fit space). The columns for part 'b' of Figure 26 are the same as for part 'a' of Figure 26.

According to the one-against-all approach (Witten and Frank, 2005), on each new datasets 'Di' the decision tree algorithm is trained; resulting in 'M' trees in total. Whenever a new faulty trace comes, each decision tree predicts its category 'Ci' of the dependent variable (i.e., the faulty function) along with a probability of being faulty. A category 'Ci' (i.e., faulty function) with the highest probability is considered as the correct prediction. Empirical evidence (Polat and Gunes, 2009) shows that training multiple decision trees (one-against-all) on several binary datasets yields better results than training a single decision tree on a dataset with many categories of dependent variable.

An excerpt of the trained decision tree generated for part 'b' of Figure 26 is shown in Figure 27. Each row contains a function, its frequency of occurrence, and the name of a faulty function after a colon sign if any. Function name with the frequency value represents the node of a tree and the faulty function name after the colon sign represents the leaf of a tree. For example, the decision tree of Figure 27 shows that if in a failed trace the occurrence of the function "portspec", "adddef" and "recgrdef" is less than or equal to "0" and the function "Get1Real" is <=1, then the faulty function is "Get1Real". In other words the tree represents If-Then-Else statements. A total of 120 different decision trees (one for each faulty function, when automatically possible)[18] were

---

[17] In data mining terminology, the faulty function is a dependent variable and the function-calls are independent variables.

[18] There were a total of 136 functions in the Space program. On some faulty (mutated) functions no test cases failed or there were not sufficient statements for valid application of mutation operators for some functions (see Section 3.4.2). Similar was the case with the UNIX utilities.

generated by using the one-against-all approach for the Space program; an excerpt of one of them is shown in Figure 27.

The tree of Figure 27 was obtained by applying the J48 algorithm in the data mining tool Weka (Witten and Frank, 2005), which is an implementation of the C4.5 decision tree algorithm. The C4.5 decision tree algorithm is the most widely and practically used algorithm. It is suitable for a dataset with numerical values (e.g., see Figure 27) of independent variables, unlike ID3 decision tree algorithm (Witten and Frank, 2005) which works only with nominal values of independent variables. The details of the C4.5 algorithm can be found in standard text by Quinlan (1993) and Witten and Frank (2005). We have avoided showing here the details due to the complexity, size and cluttering of text.

Figure 27: The C4.5 decision tree model for the function "Get1Real" of the Space program from failed traces of mutants by using one-against-all approach.

Several other algorithms for classification also exist, such as neural networks, support vector machines, naïve Bayes classifiers, etc. (Witten and Frank, 2005). We chose the decision tree algorithm because during our empirical analysis other algorithms have not yielded as efficient results as the decision tree in terms of either performance or accuracy. For example, we used Weka (Witten and Frank, 2005) tool to evaluate different algorithms on the trace dataset. Naïve Bayes, using Weka, resulted in the lower accuracy

than the decision tree and the neural network took long time for training on the trace dataset. Support vector machine (SVM) with linear kernel resulted in the similar accuracy as the decision tree, but SVM was slightly slower than the decision tree using Weka. We selected the decision tree algorithm because of its simplicity, wide use, easy to understand rules, accuracy and the speed. Nonetheless, the purpose of this research is not the comparison of the classification algorithms, but to provide evidence that the classification algorithms are useful in fault localization. The formal comparison of these algorithms is out of the scope of this paper, and any classification algorithm can in fact be used.

Following the training of the decision trees on mutant-traces, actual traces were provided as input to the decision trees for prediction. Each decision tree, generated using the one-against-all approach, predicted a faulty function with a probability. The probability of prediction in the C4.5 algorithm is determined by measuring the number of training instances correctly classified at a leaf and dividing it by the total number of instances (correct and incorrect) reached that leaf (Quinlan, 1993).

When predicting faulty functions using the one-against-all approach we made a minor modification; i.e., instead of selecting a predicted faulty function with the highest probability, we ranked the predicted faulty functions in the decreasing order of their predicted probabilities. The reason is that: (a) the developer gets multiple options if function with the highest probability is not the actual faulty function, (b) the developer's effort could be quantified using a metric to estimate effort in discovering a fault (Jones and Harrold, 2005; Di Fatta et al., 2006) (e.g., percentage of code reviewed in discovering faulty functions). The function list is then presented to the developer with the intuition that higher the function is in the list more likely it is to be faulty compared to the lower ones in the list.

An example of a ranked list of faulty functions for two different traces of actual faulty versions of the Space program is shown in Figure 28. Figure 28 first shows ranking for the trace "t1915" for version 11 of the Space program according to probabilities predicted by the decision trees. In version 11 "mksnode" is the faulty function ranked at position 2.

Similarly, for the trace "t4455" of version 4 the function "Get1Real" is the faulty function.

	Function	Probability
Space_V11_t1915		
Rank 1	intmin	0.044
Rank 2	mksnode	0.042
Space_V4_t4455		
Rank 1	circspec	0.119
Rank 2	prnfile	0.0166
Rank 3	Get1Real	0.0165

Figure 28: Ranking of suspected faulty functions in real failed traces obtained from the decision tree model of failed traces of mutants.

Finally, as actual failed traces arrive and faulty function is identified in those traces, the actual traces can then be added to the failed trace collection of mutants. If there are failed traces of faulty functions of prior faults then they can also be added to the initial collection of mutants. The decision tree can then be re-trained based on the updated collection to diagnose future failed traces (see Figure 24). If 50-90% (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) of the field failures are rediscoveries of previous faults then the refined knowledge of failed traces will actually improve the accuracy of discovery of faulty functions in the new actual failed traces.

3.5 The F007-plus Strategy

Recall that F007-plus uses mutation to facilitate F007-basic in discovering new and old faulty functions. F007-plus actually trains decision trees on the failed traces of mutants and failed traces of prior faults. Following are the steps of F007-plus, which are further explained in the following sections:

1. Measure code metrics (e.g., executable LOC, cyclomatic complexity, maximum nesting, ratio of comments to LOC) of every function of current and prior releases. Measurement of metrics facilitate F007-plus in identifying the characteristics of "faulty" and "not-faulty" functions, which F007-plus can use to predict suspected functions in future release.

2. Train the decision trees on these code metrics, using the cost sensitive machine learning[19] of prior releases and identify the suspected functions (to be faulty) in a current release. The identification of suspected functions is important because only 20% of the code is responsible for 80-100% of faults (Gittens et al., 2005; Ostrand et al., 2005) and we can save time of mutant traces collection by generating mutants of approximately 20% suspected functions.

3. Generate mutants of the suspected function identified in step 2 and collect mutant traces of those suspected functions. In this step we actually prepare a collection of failed (mutant) traces of the suspected functions that makes a training-set for the decision tree as explained in Section 3.4.3.

4. Train the decision trees, as explained in Section 3.4.3, on the traces of mutants and prior faults, and identify faulty functions in the traces of a current release. This step is adopted from F007-basic with the addition of mutant traces, which facilitates in predicting new and old faulty functions in a failed trace.

3.5.1 Step 1: Measuring the code metrics of functions

In Section 3.4.2, we mentioned that time and effort to collect mutant traces can be reduced by collecting ten failed traces per mutant and by selecting only three mutants per function. Literature indicates that 20% of the code is responsible for 80-100% of the faults (Gittens et al., 2005; Ostrand et al., 2005). This means we can save further time of mutant-trace collection by generating mutants of only the suspected faulty functions.

The suspected faulty functions in the current release can be identified using the code metrics (e.g., cyclomatic complexity, path counts, executable lines of code, coupling, inheritance depth etc.) based techniques proposed in the literature for software components (Basili et al., 1997; So et al., 2002) and classes (Emam et al., 2001) (method ‘c’). For example: predicting faulty components by building the C4.5 decision tree model

[19] In cost sensitive learning, a classifier (e.g., decision tree) is forced to make lesser error on one type of category (e.g., faulty functions) and more errors on other type of category (e.g., not-faulty functions).

of source code metrics of the past releases (Basili et al., 1997); using fuzzy logic based model of the inspection data of the past releases to predict faulty components in the current release (So et al., 2002); using logistic regression on object oriented code metrics to identify faulty classes (Emam et al., 2001); and application of different machine learning algorithms via code metrics to classify as high or low maintenance cost, reusability and fault proneness of software components (Lounis and Ait-Mehedine, 2004). Below we describe how F007-plus uses source code metrics to predict the suspected functions (to be faulty) in a similar manner as described in the prior techniques for components (Basili et al., 1997; So et al., 2002) and classes (Emam et al., 2001); however, empirical evaluation and a formal comparison of these techniques (Basili et al., 1997; So et al., 2002; Emam et al., 2001) is out of the scope of this paper.

In F007-plus we measured four source code metrics for every function of each release, such as:

- **Executable lines of code**: the lines of code of a function excluding comments;
- **Cyclomatic complexity**: the amount of decision logic of a function;
- **Max nesting**: the total nesting level of control constructs (if, while, etc.) in the functions; and
- **The ratio of comment to the code**: the ratio of commented lines to the executables lines of a function.

We chose the above four source code metrics in F007-plus because they yielded the best results in our case. We actually used the Understand tool[20] to extract the source code metrics of functions. Using this tool we were able to extract the following source code metrics: declarative statements, lines of code, comment lines, executable statements, cyclomatic complexity, maximum nesting, path count, ratio of comment to code, blank lines, and few variations of cyclomatic complexity. We chose the above four source code

[20]Code metrics were extracted using the Understand tool (www.scitools.com).

metrics because the use of other code metrics (extracted using the Understand tool) with the decision tree resulted in lower accuracy of predicting suspected functions in a new release. Also these code metrics were found useful in the literature (Basili et al., 1997; So et al., 2002; Emam et al., 2001). Many other code metrics exist (e.g., coupling, fan-in, fan-out, Halstead complexity, etc.) and can be extracted using different tools. However, as mentioned earlier the focus of this paper is not the evaluation of code metrics, but to show that we can leverage the code metrics based techniques to achieve the goal of identifying faulty functions in actual traces. Thus, other code metrics which yield better results can be used too.

Moreover, if a function was found faulty in a release then we assigned it a label of "faulty", otherwise we assigned it a label of "not-faulty". In data mining terminology, independent variables are the code metrics and a dependent variable is the fault proneness of the function; i.e., "faulty" or "not-faulty".

For example, consider part 'a' of Figure 26. In this case, the code metrics formed the columns of Figure 26, each row would represent the values of the four code metrics for each function of a release and the last column would contain "faulty" or "not-faulty" status of the function in a release.

3.5.2 Step 2: Using the decision tree on the code metrics

Secondly, we trained the decision tree on the code metrics of functions of the preceding release to predict functions in the following release as (possibly) "faulty" or "not-faulty"[21]. For example, we trained the decision tree on the code metrics of the functions of the release 1 of the "Flex" program to predict faulty functions in release 2 of the "Flex" program. Similarly, we used release 1 and 2 to predict the possible faulty functions in release 3. In general, we trained the decision tree on the code metrics of releases 1 to n-1 to identify the possible faulty functions (or suspected functions) in release n.

---

[21] We trained a single decision tree in this case because there are only two categories of dependent variable: faulty and not-faulty.

The above approach of training the decision tree to predict faulty functions result in high accuracy with few errors (incorrect predictions) of "faulty" and "not-faulty" functions For example, usually 20% of the functions in a program are "faulty" and the rest (majority) of the functions are "not-faulty" and a decision tree classifier gets biased towards "not-faulty" functions. This biased decision tree classifier would predict suspected functions with approximatley 90% accuracy with around 10% of "faulty" functions predicted as "not-faulty" and about 1-2% of "not-faulty" functions predicted as "faulty".

In this case, however, the cost of predicting a "faulty" function as "not-faulty" (false negative) is much more than the cost of predicting a "not-faulty" function as "faulty" (false positive). This is because if a function is not identified as "faulty" for the release of a program then we can not generate mutants for that function. If there is no mutant, then there will be no failed traces for the faulty function and the faulty function cannot be predicted in the actual failed trace. On the other hand, a few extra false positives will result in the generation of mutants of few more "non-faulty" functions and will not adversely affect the accuracy of prediction of faulty functions in failed traces. For example, if we get around 70% accurate predictions of (suspected) faulty and not-faulty functions from the decision tree with hardly any "false negatives" and about 30% "false positives" then we can generate mutant traces of all the (to be) faulty functions and use those traces to identify faulty functions in new traces; however, if we don't have mutant traces of the suspected functions we cannot identify those functions in new failed traces.

In short, we use the cost-sensitive learning strategy (Ting, 2002; Witten and Frank, 2005) to train the decision tree on the code metrics. Costs in cost sensitive learning are the values which force the decision tree to make lesser error on one type of predictions (e.g., faulty) than the other (i.e., not-faulty). Suppose 'C_f' is the cost of misclassifying a function as "faulty" and 'C_{nf}' is the cost of misclassifying a function as "not-faulty". Training instances belonging to the "faulty" category are assigned weights according to the cost 'C_{nf}' and the training instances belonging to the "not-faulty" category are

assigned weights according to the cost 'C_f'[22] . The decision tree is then trained with the normal procedure on the training set except that new weights of instances are used instead of normal unit weights of instances (Ting, 2002).

For example, if we set 'C_f' to 1 and 'C_{nf}' to 20 then it means that the cost of misclassifying a function as "not-faulty" is 20 times more than misclassifying it as "faulty". Hence, the weights of instances in the training set belonging to a faulty class will be 20 times more than the instances of "not-faulty" class.

Thus, in this step, we first generate a training set with the high misclassification cost of functions as "non-faulty" ('C_{nf}') and the low misclassification cost of the functions as "faulty" ('C_f'). The selection of the cost ratios depend on the subjective judgment of the user of a particular problem (Witten and Frank, 2005). We developed our own criteria for selecting the cost values: (a) we selected those cost values on which approximately 70% of faulty functions were correctly predicted as faulty in the training-set (prior releases); and (b) we used the training set of these identified cost values to predict expected faulty functions in the test-set (current release) using the decision tree. We selected the threshold value of 70% for training-sets because at this level we found that the majority of faulty functions in test-sets were correctly identified with fewer false negatives (incorrect not-faulty predictions) and not many false positives.

For example, if 20 functions are faulty in a training set and 80 functions are not-faulty, then we select those cost values at which 12-15 faulty functions are correctly predicted as faulty in a training set. This will also result in 10-30 not-faulty functions predicted as faulty (false positives). Overall there will be few more suspected (to be faulty) functions (i.e., including true positive and false positives) but fewer faulty functions incorrectly predicted as not-faulty. Note that, correct prediction of 12-15 faulty functions is approximately 70% but not exactly 70%. We choose cost ratios such that this value remains around 70% because sometimes selection of two adjacent cost values can make

---

[22] Actual equation to measure the weights using cost can be found in Kai Ming Ting's paper (Ting, 2002). We used the "cost sensitive learning" algorithm in the Weka API (Witten and Frank, 2005) which implements Kai Ming Ting's technique to make any classifier cost sensitive.

all the functions or the majority of the functions predicted as faulty (e.g., [C_f =1, C_{nf} =30] and [C_f =1, C_{nf} =40] can have such an effect and selecting any other cost value in between them could result in the same predictions as [C_f =1, C_{nf} =30]). Thus, we select the cost values by setting the threshold of around 70% for correct suspected (to be faulty) functions predictions for a training set, such that only a small proportion of false positives are predicted (i.e., not-faulty predicted as faulty). This criterion will help maintainers in identifying the suitable cost ratios for their software systems. An example execution of F007-plus is shown in Section 3.5.5.

3.5.3 Step 3: Generating mutants of the suspected functions

In the third step, we generate (three) mutants of the suspected faulty functions and collect failed traces on those mutants. See Section 3.4.2 for the mutant generation process.

3.5.4 Step 4: Identifying faulty functions in the traces of the current release

In the last step, we train the decision tree algorithm, as explained in Section 3.4.3, on failed traces of mutants collected in Step 3 and failed traces of prior releases. The trained decision tree is then used to discover faulty functions in the traces of the failures of the succeeding software release.

3.5.5 Executing F007-plus

In Table 12, we show the cost ratio $C_f : C_{nf}$ that we used for the different releases of the UNIX utilities. For example, the value 1:20 ($C_f : C_{nf}$) for "release 2" of the program Flex shows that we set C_{nf} to 20 and C_f to 1 on the training set obtained from release 1 to estimate faulty functions in release 2. Similarly, ($C_f : C_{nf}$) 1:5 in the last column for the Flex program demonstrates that we set this value on the training set[23] obtained from release 1 to 4 to estimate faulty functions in the release 5 of the Flex program.

---

[23] We actually used SQL queries with "UNION" keyword when extracting code metrics of multiple releases. This means in the training set of release 1 to n-1 more than one record for a function would exist only if a function was changed in any one of the relase from 1 to n-1;otherwise, only one record per function would be present in the training set.

Table 12: Misclassification cost ratio "C_f : C_{nf} " for the following releases of the UNIX utilities using training-set of previous releases.

Program	Release 2	Release 3	Release 4	Release 5
Flex	1:20	1:30	1:5	1:5
Grep	1:45	1:10	1:5	NA
Gzip	1:5	1:7	1:3	NA
Sed	1:110	1:10	1:85	1:45

Recall from Section 3.5.2, the selection of cost ratios depend on the subjective judgment of the user of a particular problem (Witten and Frank, 2005). We selected the cost ratios when approximately 70% of the "faulty" functions in the "training-set" were correctly classified as described in Section 3.5.2. That is for all the programs and releases, we developed a criterion that: if 70% of the faulty functions are correctly predicted as faulty in the training set with a small proportion of incorrectly predicted not-faulty functions as faulty, then we select those cost ratios for the test set. For example, consider release 3 (R3) of the Flex program in Table 12, where we selected the cost ratio of (C_f : C_{nf}) 1:30. The steps of F007-plus on R3 of Flex are described below:

- We selected this cost ratio of 1:30 for R3 of Flex because in the training set (of R1 and R2) 20 functions out of 26 functions were correctly classified as "faulty" (i.e., approximately 77% of the faulty functions were correctly predicted in the "training-set"), and 110 functions out of 186 were correctly classified as "not-faulty" on those cost ratios.

- We then assigned weights according to C_{nf} =30 to the faulty instances in the training-set and according to C_f =1 to the non-faulty instances in the training-set. This resulted into the new cost sensitive training-set.

- We generated the decision tree from this cost-sensitive training-set of R1 and R2 to predict suspected "faulty" functions in the test set of release 3 (R3). This decision tree predicted 8.0 out of 12.0 functions correctly as "faulty", and 93 out of 152 functions correctly as "not-faulty".

- We generated mutants of 67 "faulty" functions—i.e., 8.0 correctly predicted faulty functions and 59 (152-93) incorrectly predicted faulty functions—for the release 3 of the "Flex" program.

- Finally, mutant traces were collected on mutants of 67 faulty functions (as described in Section 3.4.2), and the decision tree is generated (as described in Section 3.4.3) from those mutant traces and failed traces of prior releases R1 and R2. This decision tree then predicted faulty functions in the actual traces of release R3. The accuracy of prediction of faulty functions in actual traces is shown in Section 3.8.

Following this approach of F007-plus, we also performed experiments on all other releases and other programs (i.e., Flex, Grep, Gzip and Sed) using the cost ratios shown in Table 12.

An ultimate measure of performance of a cost sensitive learning algorithm is average misclassification cost of testing examples (or traces in test-sets in our case). A specific threshold doesn't exist but a cost sensitive learning algorithm should have a low average mislcassification cost. The average misclassification cost is measured by using Equation 3. In Figure 29, we show the average misclassification cost for different releases of each of the four programs: Flex, Grep, Gzip, and Sed. In Figure 29, Y-axis shows the average misclassification cost, and X-axis shows program releases such that earlier releases form training-sets (of code metrics) for F007-plus and succeeding releases form test-sets. Each point on the series represents the average misclassification cost corresponding to the cost ratios in Table 12 for each program. For example, first point on the "Flex" series in Figure 29 show that when F007-plus was trained on the code metrics of release 1 and predicted suspected functions in release 2 using the cost ratios 1:20 then the average misclassification cost was approximately 0.6. Note that the cost ratios are different for every release of a program, but the criterion of setting those cost ratios is the same (i.e., an approximate 70% threshold).

$$Average\ misclassification\ cost = \frac{FP * C_f + FN * C_{nf}}{N}$$

Equation 3: Measures the average misclassification cost where: FP is total functions predicted as false positive, C_f is the misclassification cost of predicting a function as faulty, FN is total functions predicted as false negative, C_{nf} is the cost of misclassifying a function as not-faulty, and N is the total number of traces.

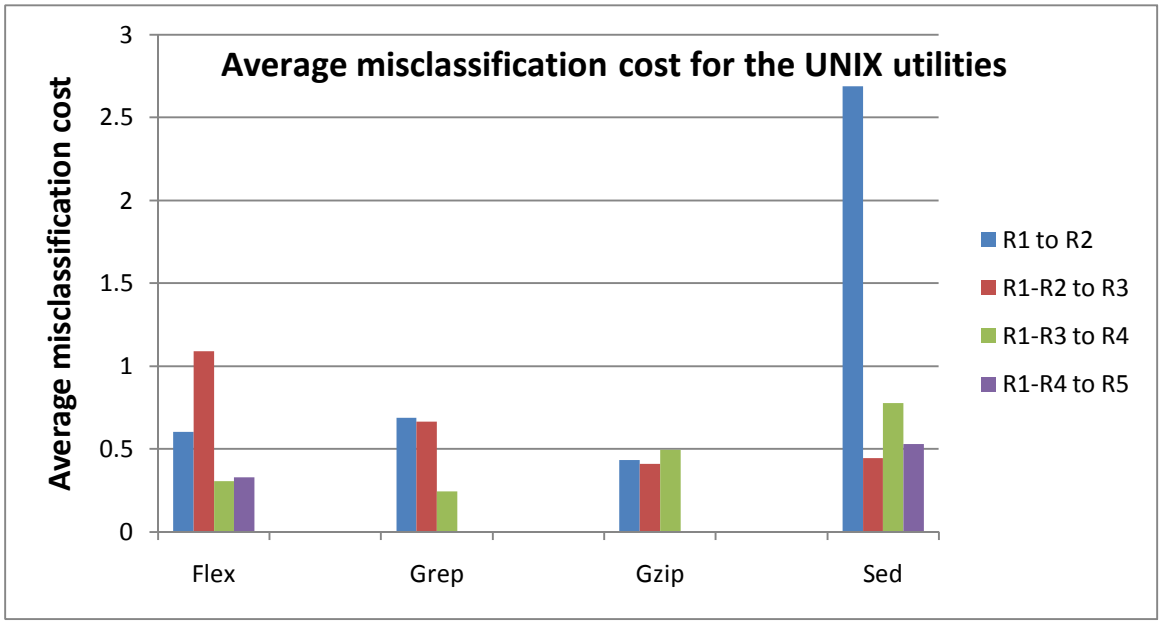


Figure 29: Average misclassification cost for the UNIX utilities.

It can be observed from Figure 29 that mostly the average misclassification cost deceases as the number of releases increase, or in other words as the number of training instances increase the average misclassification cost goes down. However, in some cases the average misclassification also increases slightly. The reason is that we selected different cost ratios for each release of a program. Usually in the cost sensitive learning, cost ratios are kept same. In our case we have delveoped a criterion (of 70% threshold) to select cost ratios and this crierion remains constant. The reason for developing such a criterion is that every program is different, and setting of different cost values by maintainers is not straight forward even if they have the knowledge of cost sensitive learning. Thus, we

selected the same criterion of 70% threshold for true positives in the training set with a small proportion of false positives. In short, in our case the cost ratios may not be the same but the criterion for selecting those cost ratios is the same.

Overall, the average misclassification cost in Figure 29 remains low or decreases (mostly) over number of releases. This means that the number of false negatives and false positives would approach to zero if the average misclassification cost approaches zero. Also, if there are fewer false negatives (FN) then the average misclassification cost will be high because we have high 'C_{nf}' value (see Equation 3). This implies that F007-plus predicts fewer false negatives and false positives, even if the training set contains instances from the first release. The suspected (to be faulty) functions predicted by F007-plus constituted approximately 10-40% of the total functions. For example F007-plus predicted: (a) 44-56 faulty functions in the five releases of the Flex program; (b) 12-43 faulty functions in the four releases of the Grep program; (c) 29-45 faulty functions in the four releases of the Gzip program; and (d) 4-28 faulty functions in the five releases of the Sed program. After identifying the susepcted functions, we collected mutant traces for those functions and trained the decision trees on those traces. The results showing the accuracy of prediction of faulty functions in actual traces are described in Section 3.8.

3.6 Implementation, Scalability and Runtime Performance of F007-plus

In this section, first we provide details of the implementation of F007-plus in Section 3.6.1. Second (in Section 3.6.2) we discus the issue of scalability of F007-plus on large programs with millions of lines of code. Thirdly, we show the execution time of F007-plus.

3.6.1 Implementation Details

We implemented F007-plus as a Java application in NetBeans, and used MySQL database to store processed traces (e.g., functions and occurrences). We optimized F007-plus for bulk reads of large traces from hard disk, bulk inserts of large records into the database, and used different table-indexes in MySQL database. We also used SWI Prolog based mutant generation tool developed by Andrews et al. (2005). The tool (Andrews et

al., 2005) generates mutants for almost every statement of the program, which results in a large number of mutants. For example, it generates 12262 mutants for the Space program. We modified this tool to generate three random mutants for every function of a program: (a) first we developed a program in Java (using regular expressions) that extracted functions and their locations from C program; and (b) second we randomly selected three mutants (generated by the tool from Andrews et al. (2005)) for one of the lines within a function.

Moreover, we downloaded (Do et al., 2005) the UNIX utilities (i.e., Flex, Grep, Gzip and Sed) and the Space program with several UNIX based scripts to enable faults in different releases of programs and run test cases on those faults of a particular program release. We also modified those scripts to automatically compile mutants in functions of programs, and to automatically run test cases on mutants of functions. Recall, from Section 3.4, we collected execution traces using Etrace (2008) and collected up to 10 failed traces per mutant. Using the scripts for mutants (and Etrace), we collected (failed) mutant traces, and using the original downloaded scripts (and Etrace) for the UNIX utilities we collected actual failed traces.

We generated mutants and collected (actual and mutant) failed traces on Ubuntu 10.02 on a 3GHz CPU with 3 GB of RAM. We executed F007-plus (i.e., storing traces into database, generating decision tree from the traces in database, and predicting faulty functions in the actual traces) on Windows 2008 server on dual core 2.5 GHz CPUs and 4GB of RAM.

3.6.2 Scalability

In this paper, we have experimented on only medium size commercial or production level programs. However, the steps of F007-plus strategy are scalable to large programs with millions of lines of code. For example, in F007-plus we identify suspected faulty functions in a current software release, which would be equivalent to approximately 20% of functions of a program—if 20% of the code is responsible for 80% of the faults (Gittens et al., 2005; Ostrand et al., 2005). In our execution of F007-plus on the UNIX utilities, F007-plus resulted into about 10-40% of suspected functions (see Section 3.5.5).

Further, we selected only three mutants per functions and generated only 10 failed traces per mutants to keep the resource and time consumption to minimum. Also, failed traces of in-house testing can be used as well alongside mutant traces and failed traces of previous releases because: (a) from our earlier study we have found an overlap in the location of field and in-house faults (Gittens et al., 2005); and (b) same 20% of the code is responsible for 80% of the faults. Thus, the decisions taken during the design of F007-plus make it scalable to large programs.

3.6.3 Execution Time

The execution time to generate mutants from the modified tool for the subject programs is shown in Table 13. Table 13 shows the name of a program, minimum-maximum lines of code for the different releases of a program, minimum-maximum mutation time for the different releases, and the minimum-maximum number of mutants for the different releases. For example, the modified mutation tool took 21-44 seconds for 320-517 mutants of the Sed program with 4711-9266 LOC.

In Table 14, we show the average size of an actual trace, average size of a mutant trace, and I/O time per trace. I/O time per trace includes time to extract function-calls and their occurrences from a failed trace along with the storage into MySQL database. It should be noted that this processing time is required to be done once for a collection of failed traces.

Table 13: Mutation time for the subject programs.

Program	LOC	Mutation Time (sec)	Mutants
Flex	8250-9831	64 -251	395-477
Grep	8484-9041	39-60	394-425
Gzip	4032-5103	23-31	246-319
Sed	4711-9226	21-44	320-517
Space	6218	45	361

In addition to this time, there was time required to generate the C4.5 decision tree model, which was dependent on Weka (Witten and Frank, 2005) API implementation. The maximum time for the C4.5 tree generation from mutants and evaluation on actual 71958 traces was approximately 15 minutes for the “Space” program. In practice, the decision

tree model is also required to be generated once, and only needs to be updated when traces with new faulty functions are included in the database. The only regular work required is to predict a faulty function in a new failed trace, which consumes few seconds.

Table 14: Processing time for traces.

Program	Avg. size of an actual trace (KBs)	Avg. size of a mutant trace	Time per trace with (I/O) sec
Flex	440.19	595.16	1.809
Grep	483.34	1321.52	0.431
Gzip	600.83	315.299	4.047
Sed	57.79	150.65	0.369
Space	33.62	48.2	0.235

3.7 Case Studies to Investigate Research Questions: (Q1) Similarity of Traces among Faulty Functions and (Q2) Discovering Actual Faults using Mutant Faults

In this section, we investigate the answers to the following questions posed in Section 3.1: *(Q1) Are the function-call traces of some faulty functions similar and the function-call traces of some faulty functions different? (Q2) Can the mutant faults be used to discover actual faults?* The answers are sought using the F007-plus technique described in Section 3.3.

In Section 3.7.1, we determine the answers to these questions by making every function faulty using mutants. In Section 3.7.2, we further investigate the questions by making only the selected functions faulty. Finally, Section 3.7.3 summarizes this section by mapping questions to the findings.

3.7.1 Making every function faulty using mutants to identify faulty functions in actual traces

Recall from Section 3.4.2 that we randomly selected three mutants per function and collected ten failed traces per mutant; that is, a maximum of 30 failed traces for every function of a program. During our investigations, we observed that for some functions the number of failed traces per function were less than the maximum limit of 30. This is because, sometimes, the randomly selected mutants did not compile, a few test cases

failed on the mutant, or no test cases failed at all on the mutant. In short, there were 30 or less failed mutant traces per function.

Nonetheless, in order to investigate how many failed traces of mutants per function are enough to identify faulty functions in the actual failed traces, we experimented with a maximum of 5, 10, 15, 20, 25 and 30 mutant traces per function. In other words, we trained the decision tree on 5, 10, 15, 20, 25 or 30 failed traces to identify faulty functions in the actual traces. Figure 30 shows the results obtained for 5, 10, 15, 20, 25 and 30 mutant traces per function for the Space program. In Figure 30, the X-axis represents the percentage of the program to be examined in discovering faulty functions. It is measured by the percentage of functions[24] reviewed up to the discovery of faulty functions in a program, as shown in Equation 4.

$$\left[\begin{array}{c} \%\ of\ \ program \\ to\ review \end{array}\right] = \frac{Functions\ reviewed\ upto\ the\ faulty\ function}{Total\ \ functions} * 100$$

Equation 4: Estimating program review effort in functions.

Using Equation 4 we compute a score for each failed trace as the percentage of a program (i.e., functions or statements) need to be reviewed to find the faulty function. Horizontal axis (X-axis) represents the percentage of program that needs to be examined and is divided into segments. Each segment is 10 percentage points except for the first 10 segments which are divided into 1 percentage points; i.e., 1-10% segments are divided into 1 percentage points and 90-100% segments are divided into 10 percentage points each. Vertical axis (Y-axis) measures the cumulative percentage of failed traces that achieve a score within a segment[25]. For example, in part 'a' of Figure 30, the point (10, 60) on a series "using 25 traces per function" (i.e., marked by "▬" shows that faulty

[24] We used functions as the programmer would review the functions in the function-call trace to discover the faulty functions, not statements.

[25] We have taken this approach from the similar graphical convention used for evaluation of the developer's effort by Jones and Harrold (2005), Wong et al. (2007) and Di Fatta et al. (2006).

functions in approximately 60% of the actual failed traces were discovered by reviewing 10% or less of the code (functions) for the Space program. This identification in the actual traces was done by training the decision tree algorithm on 25 or lesser failed traces

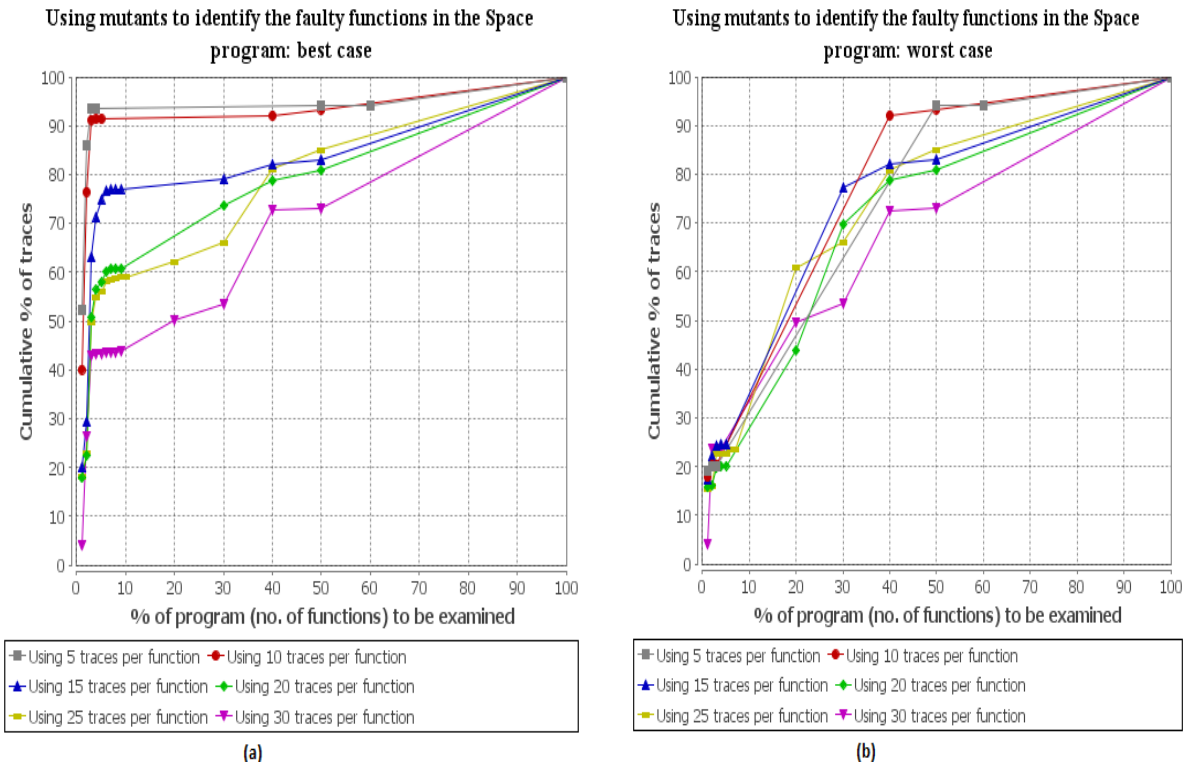


Figure 30: Faulty function prediction accuracy for the Space program on its failed traces of actual faults using the failed traces of mutants of all functions.

of mutants of every function of the Space program. Note that, straight lines at the end of a series till the 100% traces when there are no more points visibile on a series mean that: F007-plus does not result in any more predictions of faulty functions in traces and a developer identifies faulty functions by random guesses till the 100% traces. For example, in the case of the series “using 25 traces per function” in part ‘a’ of Figure 30, 84% of the failed traces were resolved correctly by reviewing 50% of the program using F007-plus after which a developer randomly gusesses the faulty functions in the remaining 16% of traces.

Recall from Section 3.4.3 that F007-plus generates a ranked list of suspected functions for a potential failed trace. It is possible that F007-plus can list two or more functions at

the same rank, then the best case effort entails that the first function to be examined is faulty and the worst case effort entails that the last function to be examined is faulty. For example, suppose there is one function listed at rank 1, and five functions listed at rank 2. The best case effort is that the faulty function is the second function to be examined (i.e., one at rank 1 and one at rank 2), whereas the worst case is that the faulty function is the sixth to be examined.

In Figure 30 part 'a' shows the best case effort of the programmer by using F007-plus with 5-30 mutant traces per function of the Space program, and part 'b' of Figure 30 shows the worst case effort of the programmer with F007-plus by using the same number of mutant traces. In Figure 30, the same series is represented by the same colour and symbol in both part 'a' and 'b'. For example, "using 25 traces per function" series is represented by the pink colour and the symbol "▬" in both the worst and the best case.

It can be observed from Figure 30 (part 'a' and 'b') that when F007-plus uses fewer mutant traces per functions then the best case effort is higher than larger number of mutant traces per function; whereas, the worst case effort is lower or similar to larger number of mutant traces per function. For example, when five mutant traces per functions were used then F007-plus identified faulty functions in 90% of the traces on the review of 3% or lesser program in the best case (see part 'a'); whereas in the worst case (see part 'b). F007-plus identified faulty functions in only 20% of the failed traces using five mutant traces per function. If the difference between the worst case and the best case is too high for a series then it means most of the functions are listed at the same rank, and the use of particular numbers of mutants per functions represented by that series is ineffective. This also means that using fewer mutant traces, the decision tree was not able to get sufficient information to predict faulty functions in actual traces, and most of the suspected faulty functions were predicted with the same probability.

In the case of Figure 30, the use of 25 and 30 traces per functions series have a small gap between their worst and best cases, respectively: implying that there are fewer functions listed at the same rank for 25 and 30 mutant traces per function. In the case of 25 mutant traces per function, both the worst case effort and the best case effort are better than the

30 mutant traces per function. However, the gap between the worst and the best of 30 mutant traces per functions is smaller than 25 mutant traces per function. Thus, we can say that F007-plus with 25-30 mutant traces per function is able to identify faulty functions in 50-60% of the actual failed traces on the review of 20% of the code for the Space program.

The test suites in the Space program were more extensive than the ones would usually be produced in practice; that is, approximately 13,000 test cases for approximately 6000 lines of code. This means almost all of the functions and control flow paths were exercised by the test cases. However, in the UNIX utilities, the sizes of the programs were almost the same as the Space program, but the test cases were not as extensive as the Space program. The test suites of the UNIX utilities mimic the real world scenario closely. On the other hand, recall from Section 3.4.2, if the test suite does not exercise all the paths then this shows the weakness of the test suite in detecting faults, and, eventually, will leave many mutants live or equivalent.

Nonetheless, we show separately in Figure 31, the accuracy of identification of faulty functions on the four UNIX utilities (i.e., Flex, Grep, Gzip and Sed) using 5, 10, 15, 20, 25 and 30 failed mutant traces. We have randomly chosen release 1 (R1) (see Table 11) from several releases of the UNIX utilities for this experiment as manifested in Figure 31. We generated mutants of release 1 (R1) of each of the Flex, Grep, Gzip and Sed programs. We again randomly selected three mutants per function of a program and collected traces by running the test cases of the respective programs on their mutants.

In Figure 31, each series actually shows the accuracy of the identification of faulty functions on the release 1 (R1) of all of the four UNIX utilities we studied. The percentage of failed traces for each series is measured by first summing the number of actual failed traces of all the four UNIX programs that achieve a (program-review) score within each segment (on X-axis) and then dividing them by the total number of failed traces of all the four programs. The results are then shown as the cumulative percentage of failed traces on Y-axis. For example, in part ‘a’ of Figure 31 the point (30, 50) on the series “using 30 (mutant) traces per function” shows that only 50% of the failed traces

were resolved correctly in the Flex, Grep, Gzip and Sed program by reviewing 30% or lesser code.

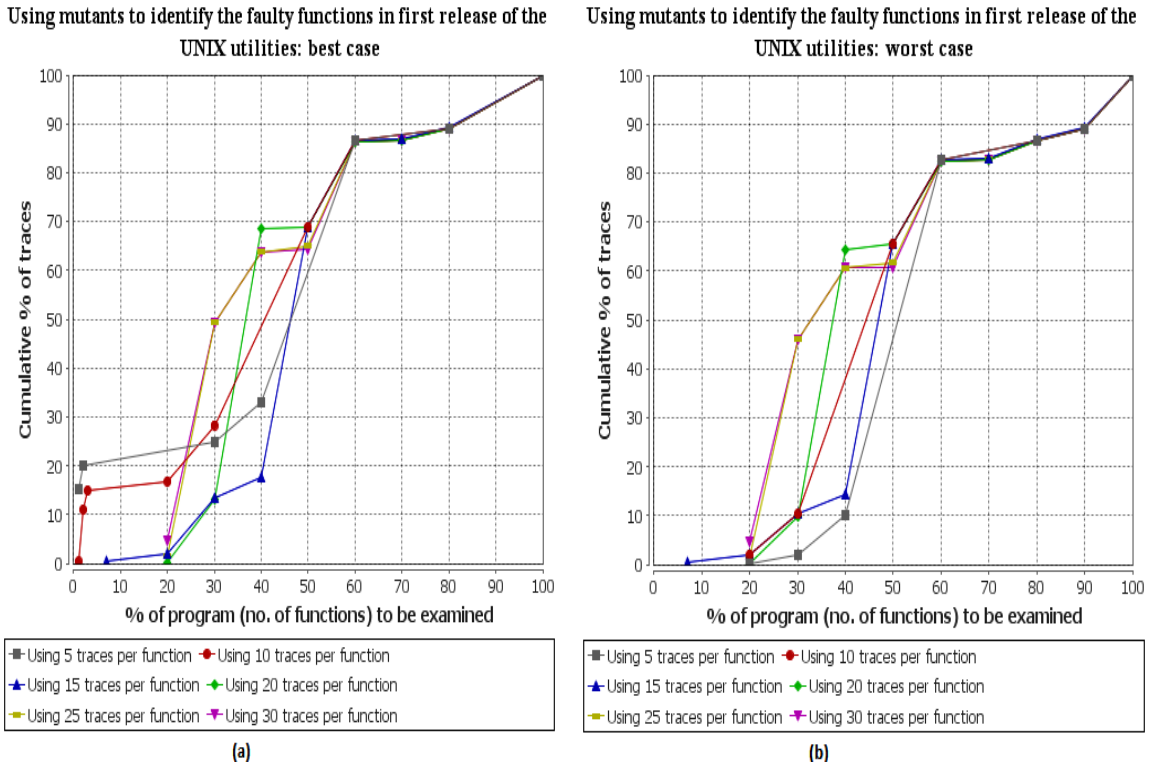


Figure 31: Accuracy of identification of faulty functions in the actual traces using mutant traces on the UNIX utilities.

In the same manner to Figure 30 (for the Space program), we also show the best case and the worst case accuracy for the UNIX utilities in Figure 31. It can be again observed from Figure 31 that the use of fewer mutant traces per function results in larger difference between the worst and the best case. Also, the use of 25 mutant traces per function results in almost identical best and worst case effort for the UNIX utilities in Figure 31. Similarly, the use of 30 mutant traces per function also results in identical best and the worst case effort. This means there was rarely more than one function listed at the same rank for 25 and 30 mutant traces per function. Thus, we can state that F007-plus with 25-30 mutant traces per function is able to identify faulty functions in 50% of the actual failed traces on the review of 30% of the code for the UNIX utilities.

Overall, the accuracy of identification of the faulty functions on the UNIX utilities in Figure 31 is low compared to the Space program in Figure 30. In the case of the UNIX utilities, the test suites were not as exhaustive as in the case of the Space program. This left many mutants live or equivalent in the UNIX utilities (see Section 3.4.2), or resulted into few failed test cases on the mutants of faulty functions. For example, in the case of the Sed program (release 1), we were able to collect the failed traces for only 87 functions out of 183 total functions. Similarly, we collected mutant traces for 79 functions out of total 89 of the Gzip program (release 1), 68 out of 142 for the Grep program (release 1), and 130 functions out of 151 for the Flex program (release 1). In the case of the Space program, which has a very large collection of the test suite, mutant traces for 116 functions[26] were collected out of total 136 functions.

This implies that extensive test suites would result in better accuracy (as in Figure 30) than shorter test suites (as in Figure 31). The reason is that extensive test suites, in our investigation, resulted in more failing traces, covering many flow paths for the faults in the same functions, and providing the decision tree more knowledge to identify the faulty functions. Further, in both Figure 30 and Figure 31, the use of "25 mutant traces per function" and "30 mutant traces per function" series shows better results than lesser number of mutant traces per functions.

The results in Figure 30 and Figure 31 show that by using the failed traces of the mutants of every function, the faulty functions in the actual trace are not entirely distinguishable, particularly for the smaller test suites. This is because 100% or closer accuracy was not obtained on the review of 1% (or little more) of the code. This also implies that different faults in the same function do not occur with the similar sequence of function calls, but overlap with the function-calls of faults in some other functions.

In fact, Figure 30 and Figure 31 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

---

[26] Some of the functions in the Space program have no statements in the body; so, no valid mutants and no actual faults were possible in them.

regularly. When a fault occurs in one of the functions of a group (e.g., M_i) then the function-calls overlap. When a fault occurs in a function in another group M_k then there are few overlapping function-calls with the function-calls of faults in groups other than M_k. The reason is that if the function-calls 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. We could still find faulty functions in 60% of the failed traces of the Space program (see Figure 30) by looking at 20% percent of the program (functions), when using 25 mutant trace per function. Similarly, we could find faulty functions in 50% of the traces of the UNIX utilities by reviewing 30% of the code (see Figure 31), when using 25-30 mutant trace per function

Thus, from these results we can state that: "A group M_i of related functions has similar function-call traces when a fault occurs in the functions of that group M_i; but the function-call traces of M_i 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 $M_i \subset N$ and $M_k \subset N$ and $N=\{functions \mid functions \in program\}$." This answers the first research question (Q1) that traces of different faulty functions are similar and traces of some faulty functions are different. Also, we found that faulty functions in 50-60% of the failed traces can be identified by making every function faulty using mutants. This identification requires the review of 20-30% of the code. This answers the second research question (Q2).

3.7.2 Making only the selected functions faulty using mutants to identify faulty functions in the traces of actual faults

In order to further validate the above proposition, we trained the decision tree on the mutant traces of only those faulty functions that were also faulty in actual traces. We made this decision because: (a) this would allow us to identify whether different faults in the same function occur with the same traces; and (b) this would facilitate in further validating that traces of some faulty functions are similar. This trained decision tree on the mutant traces of the selected faulty functions was then used to identify faulty functions in the actual failed traces. The results are shown in Figure 32 by the series "using mutant traces of the same faulty functions as in the actual traces": this series

shows both the best and the worst case marked by ■ and ●. The best and the worst case efforts, however, have overlapped and no significant difference is noticeable; except for the first point, which is approximately (1,30) in the worst case and (1,32) in the best case. These results were obtained by using 30 mutant traces per function.

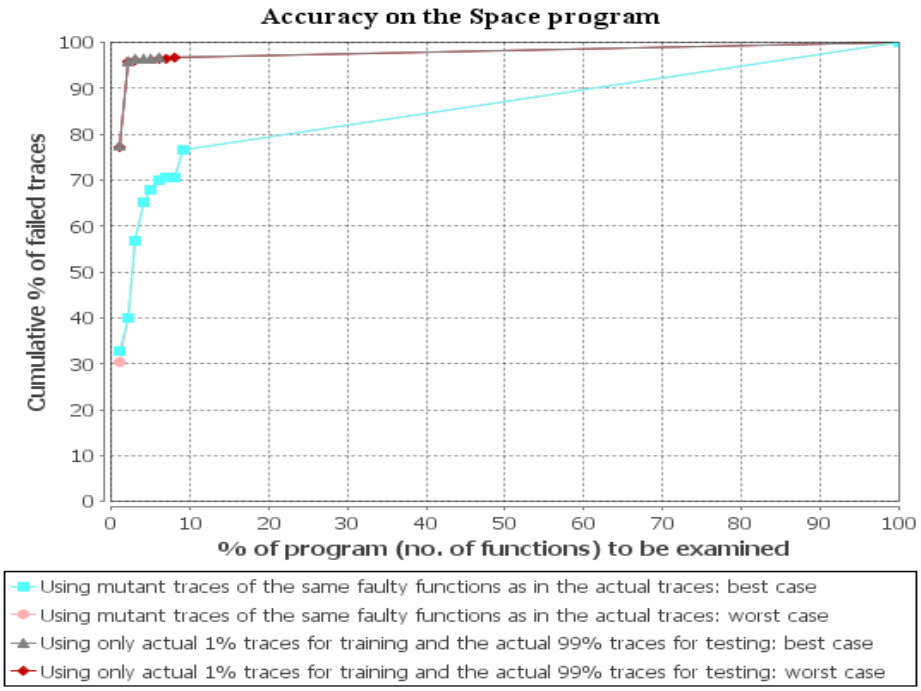


Figure 32: Faulty function prediction accuracy by using failed traces of the same faulty functions on the Space program.

In order to compare the accuracy of mutant traces of the selected faulty functions, we also trained the decision trees on 1% of the actual failed traces and tested them on the rest of the 99% actual failed traces. This 1% of the actual traces has the same faulty functions as the remaining 99% traces for the Space program[27]. This is shown in Figure 32 (for the Space program) by the best case (marked by▲) and the worst case (marked by ♦) of the series "using only 1% actual traces for training and the actual 99% traces for testing".

[27] We divided data into 100 equal parts using Weka API (Witten and Frank, 2005), each part contained equal proportion of failed traces for every faulty function. We used one part for training and the rest of the 99 parts for testing. This is called stratification; without stratification faulty functions with lesser traces would be missing from some parts-- resulting in incorrect classification accuracy.

Again the difference between the worst case and the best case is almost not noticeable. It can be observed from Figure 32 that the difference between the accuracy of identifying faulty functions using actual traces and using mutant traces is quite narrow. For example, using the mutant traces of the same faulty functions as the actual traces for the training set, the faulty functions in approximately 77% of the actual traces can be identified by reviewing 10% or less of the code. Similarly, in Figure 32, using 1% of the actual traces as the training set, the faulty functions in approximately 97% of the traces can be identified by reviewing 3% or less of the code. This shows that faulty functions in majority of the actual traces were identified by training the decision trees on different faults (mutants) of the same function.

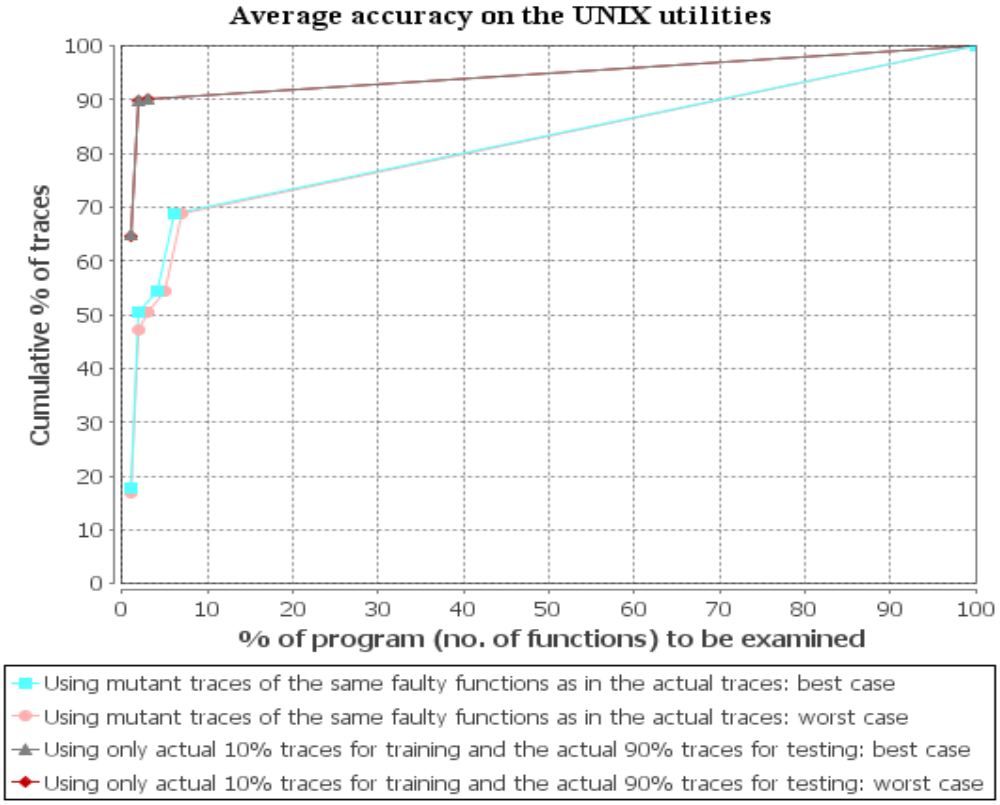


Figure 33: Faulty function prediction accuracy by using failed traces of the same faulty functions on the UNIX utilities.

Following the approach similar to Figure 32, the results on the UNIX utilities are shown in Figure 33. In the UNIX utilities we have used 10% of the actual traces for training,

instead of 1% because there were fewer failed traces in the UNIX utilities compared to the Space program (see Table 11). In the case of the Space program, there were about 72000 failed traces and the use of 10% traces still resulted into approximately 7000 traces for training. On the other hand, all the UNIX utilities had less than 500 failed traces. Due to fewer failed traces of the UNIX utilities, we selected 10% of their traces for training F007. The reason lies in the fact that the decision tree requires a sufficient number of traces for training; for example, literature (Witten and Frank, 2005) recommends selecting more than 50% of data for training when the data set is not large—the 10% we used we used is still much less than recommended 50%.

It can be observed from Figure 33 that the accuracy of "using mutant traces of the same functions as in the actual traces" for training and the accuracy of "using 10% of the actual traces" for training are quite close. Also, note that the difference between the best and the worst case efforts is hardly noticeable for the mutant traces and actual traces series. For the UNIX utilities, in Figure 33, we used 30 mutant traces per function to train the decision tree on the mutant traces (25 mutant traces per function could have been used too, as we discussed in Section 3.7.1).

In both the cases, Figure 32 and Figure 33, faulty functions in 90-95% of the failed traces can be identified by reviewing 2% (≈ 3 functions) of the program, when using a proportion of actual traces for training. On the other hand, by using mutant traces, faulty functions in approximately 60% of the failed traces can be identified by reviewing 3% (≈ 4 functions) of the program in Figure 32 and Figure 33. This implies that: (a) function-call paths triggered by different faults in the same function are not exactly the same but they are similar—because accuracy of mutants and proportion of actual faults (in Figure 32 and Figure 33) are not the same; (b) there are actually groups of related functions that occur with overlapping function-calls when a fault occurs in the functions of same group--because we need to review few functions before identifying the faulty function; and (c) different groups of functions have few overlapping function-calls, otherwise we would have had a very low accuracy of identification of faulty functions using mutants—we can identify faulty functions in approximately 70-80% of the failed traces using mutants by reviewing 10% or less of the code (in Figure 32 and Figure 33). These

observations are the same as what we observed in Figure 30 and Figure 31 in Section 3.7.1

3.7.3 Summary of the Key findings

We summarize the key findings from the case studies:

- "Related functions in a group 'M_i' have similar function-call traces when a fault occurs in the functions of that group 'M_i'; but the function-call traces of 'M_i' 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 $M_i \subset N$ and $M_k \subset N$ and $N=\{functions \mid functions \in program\}$." This answers the first research question (Q1).

- Faulty functions in 50-60% of the actual traces can be identified by reviewing 20-30% of the code by making every function artificially faulty (i.e., using mutants) and using the traces of all those mutated functions (see Figure 30 and Figure 31). This answers the second research question.

Also, the results in this section show that faulty functions in actual traces can be identified with high accuracy (60-65%) by reviewing 3% of the code using the mutant traces, provided the faulty functions in the mutant traces are the same as the actual traces (see Figure 32 and Figure 33). However, an important question is how to determine those faulty functions which are going to be faulty in actual traces? From literature, we know that 20% of the code is responsible for 80% or more of the faults (Gittens et al., 200; Ostrand et al., 2005). If we can some how identify the suspected faulty functions in the current release of a software system, train the decision trees on the mutant traces of those suspected functions then we can predict faulty functions in the actual traces. This is discussed in the next section. Therefore, this section only partially answers the question if mutant faults can be used to discover actual faults.

3.8 Evaluating F007-plus

In the previous section, we identified that if the decision tree is trained on the mutant traces with the same faulty functions as in the actual traces, then the faulty functions in the actual traces can be identified accurately. However, in reality, the actual faulty functions are not generally known and so we need to predict the probable faulty functions. This can be achieved through the method proposed in F007-plus in Section 3.5. Recall that in F007-plus we can identify the suspected faulty functions using the code metrics based cost sensitive learning (see Section 3.5), and then we can generate mutants of only those suspected faulty functions.

In the following sections, we show how F007-plus is evaluated on the four UNIX utilities (i.e., Flex, Gzip, Grep and Sed). We show the accuracy of identification of faulty functions using F007-plus in Section 3.8.1, effort in statements in discovering faulty functions in Section 3.8.2, identification of multiple faulty functions in Section 3.8.3, and the use of rules in diagnosing fault proneness of a function from the perspective of related functions in Section 3.8.4. Moreover, the evaluation of F007-plus facilitates in further investigating the question: (Q2) Can the mutant-faults be used to discover actual faults? Specifically, F007-plus would focus on how mutants can be used to accurately identify actual faults—i.e., identifying faulty functions in a trace by reviewing few functions.

3.8.1 Using F007-plus to Identify Faulty Functions in the UNIX utilities

In Figure 34, we show the average accuracy of predicting faulty functions in the actual traces of release 2, 3, 4 and 5 of the UNIX utilities. The results in Figure 34 are obtained by training F007 on the failed traces of prior releases and the failed traces of mutants of the suspected functions of a current release. In terms of mutants, we used a maximum of 30 mutant-failed-traces per function as we identified in Section 3.7 for the UNIX utilities. Recall from Section 3.4.1 and Section 3.6.1 that we collected actual failed traces on the different releases of the UNIX utilities by collecting traces of failed test cases. Also recall from Section 3.7 that 1-10% segments on X-axis are divided into one percentage points and 90-100% segments are divided into ten percentage points each.

The results in Figure 34 for the four UNIX utilities and their several releases are obtained in the following way[28]: (a) the training set contains the failed traces of the past releases and traces of mutants of the current release of one of the UNIX program; (b) the test set includes traces of a current release of the same program;(c) the score of each failed trace is again measured using Equation 4; and (d) the percentage of failed traces for a segment on X-axis is the percentage of failed traces of the same level releases --used as test-sets in above point 'b'-- that fall within each segment for the four programs of the UNIX utilities.

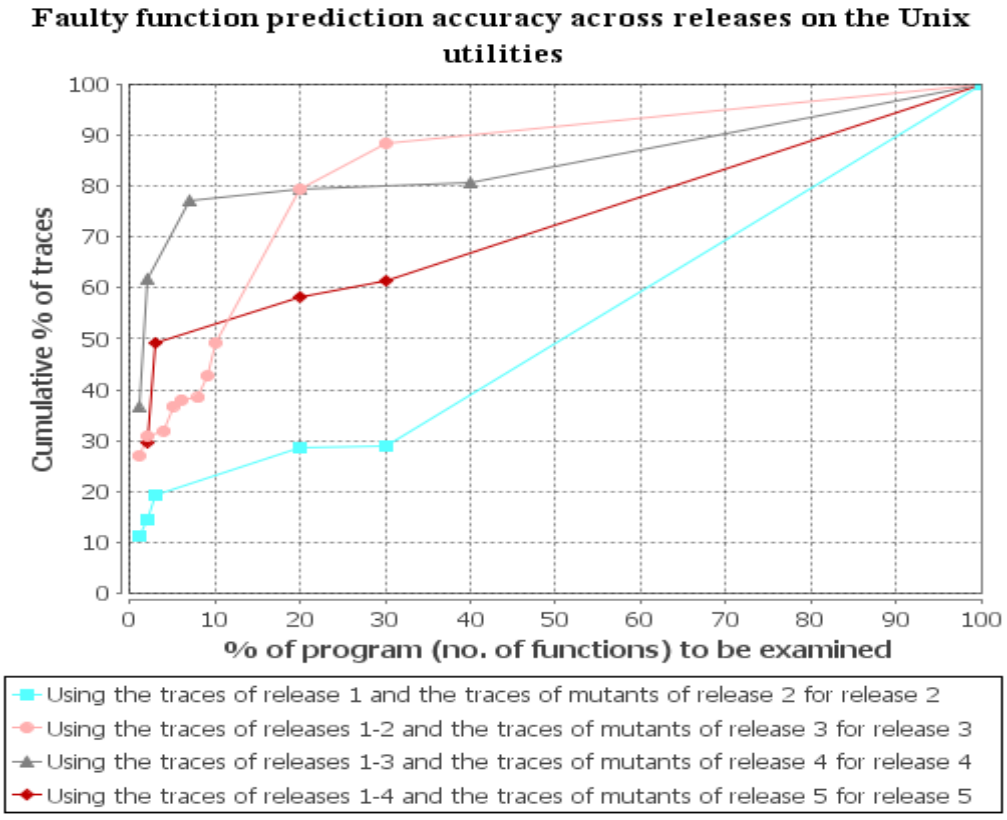


Figure 34: Faulty function prediction accuracy on the actual failed traces of the following release using the failed traces of selected mutants and the actual failed traces of the preceding release.

[28] This approach is adopted from the similar graphical convention used for evaluation of the developer's effort by other researchers (Jones and Harrold, 2005; Di Fatta et al., 2006; Wong et al., 2007).

The percentage of the failed traces for a particular segment on X-axis is measured by summing the number of actual failed traces (of the releases of four programs used as test sets) that achieve a program-review score within each segment on X-axis. Afterwards, the sum is divided by the total number of failed traces in the test release of all the four programs. For example, at the level of release 3, Flex, Gzip, Grep and Sed had 0, 173, 28 and 1 traces resolved correctly, respectively, on the program-review of 1%. The total number of failed traces for Flex, Grep, Gzip and Sed in release 3 were 362, 247,14 and 98 respectively. This resulted into an accuracy of 28% on the review of 1% of the program for release 3 of the UNIX utilities as shown in Figure 34 by the series "using the traces of release 1-2 and the traces of mutants of release 3 for release 3". For the remaining 99% segments on X-axis the results are shown as cumulative percentage. We have adopted this approach from literature as used by other researchers (Di Fatta et al., 2006; Jones and Harrold, 2005; Witten and Frank, 2005). This approach helps in avoiding cluttering of graphs, if the results were shown for all the programs separately.

Similarly, Figure 34 shows the result on other releases of the UNIX utilities by using the traces of all the preceding releases and mutants of the current release as the training set and the following releases as the test set. In Figure 34, we have only shown the best case effort to avoid cluttering the graph. Also, the difference between the best and worst case effort was not noteworthy (as in Section 3.7.2, or in some cases remained close to the best case), so we presented only one series. Similarly, in all other figures in this section we show only the best case efforts.

It can be observed from Figure 34 that faulty functions in 30-60% of the failed traces can be identified correctly by reviewing 20% of the program for release 2 and 5. Similarly, in release 3 and 4 about 80% of the failed traces can be identified by reviewing 20% of the program. The accuracy for release 2 is low compared to release 3, 4 and 5 because many mutants were not killed by the test cases (i.e., test cases did not fail on the mutants). This resulted in non-failing (mutant) traces for several faulty functions in the training set and those faulty functions were not diagnosed in the actual traces.

For example, in the case of Gzip, no test cases failed at all on the mutants of release 2 to 4. We, therefore, used the traces of mutants of release 1 of Gzip for the releases 2 to 4. In the case of other programs, the mutants (of the same faulty function) of previous releases also resulted into no failing test cases or very few mutant traces, which had no effect on the accuracy when they were included with the failed traces of mutants of the following release. The accuracy was also partially affected in some cases when some faulty functions were not estimated as faulty using the code metrics (see Section 3.5)--resulting in no mutant traces for those functions. Nevertheless, the accuracy of identification of faulty functions is still approximately 60-80% on reviewing less than 20% of the program (functions) for the majority of the releases, which is still quite significant in terms of savings in effort in reviewing functions.

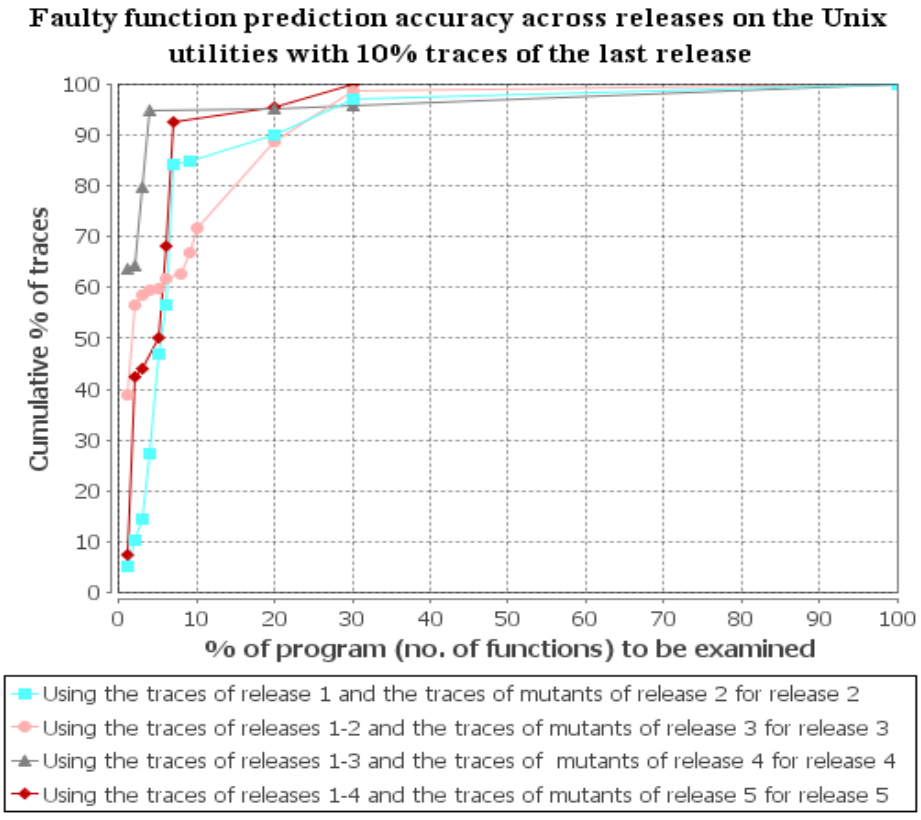


Figure 35: Faulty function prediction accuracy on the actual failed traces of the current release using the failed traces of selected mutants, actual failed traces of the preceding release, and the 10% traces of the current release.

In Figure 24, we showed that once the actual failed traces are resolved they can be added to the repository of failed traces. The decision tree can then be retrained on the collection to identify faulty functions in the actual failed traces. This is because 50-90% of the field failures are rediscoveries of previous faults (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) and using the resolved failed traces will help in improving the accuracy of identification of the faulty functions. For example, in Figure 35 we show the results by merging 10% of the actual failed traces with the actual failed traces earlier releases and the failed traces of the selected mutants. In other words, we added 10% of actual failed traces to the training sets used in Figure 34. The test set in Figure 35 is the rest (90%) of the actual failed traces for each release. It can be observed from Figure 35 that the majority of the faulty functions in the failed traces can be identified by reviewing less than 5% of the program.

The results of Figure 35 have high significance for the 50-90% rediscovered faults (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003). This is because with only limited knowledge of 10% of the actual failed traces along with the failed traces of previous releases and mutants, faulty functions in 90% of the failed traces can be correctly identified. Thus a majority of the failed traces can be correctly resolved with the minimal effort in corrective maintenance by training the decision tree on failed traces of mutants and few failed traces of actual faults. Further, even if the failed traces of the actual faulty functions are not known for the current release then we can still identify faulty functions with an accuracy of 30-80% on the review of 20% program in the failed traces of the current release (see Figure 34).

3.8.2 Measuring statements-effort in identification of faulty functions

So far, we have shown the accuracy of identification of faulty functions by using the percentage of functions reviewed as the code reviewed. However, the sizes of functions vary in a program, and functions with large sizes could account for majority of the code and hence large number faults. Therefore, in order to estimate the effort of a developer in terms of number of statements we summed all the statements of a function reviewed by a developer up till the faulty function. For example, if the fourth function in the list

generated using F007-plus was the actual faulty function then we summed the number of statements of all the four functions in estimating the effort. This is shown in Equation 5.

$$\begin{bmatrix} \%\ of\ \ program \\ to\ \ review \end{bmatrix} = \left[\frac{\sum Statements\ \ of\ \ a\ \ reviewed\ \ \ function}{Total\ \ statements} * 100 \right]$$

Equation 5: Estimation effort in statements.

In Figure 36, we show the effort of a developer in identifying faulty statements across releases of the UNIX utilities. In Figure 36, like Figure 34, the training set contained only the traces of earlier releases and selected mutants, but the effort of developer is estimated in statements. Similarly, for Figure 37, the training set contained the failed traces of earlier releases, selected mutants, and 10% of the following releases, and the effort of a developer is estimated in terms of statements.

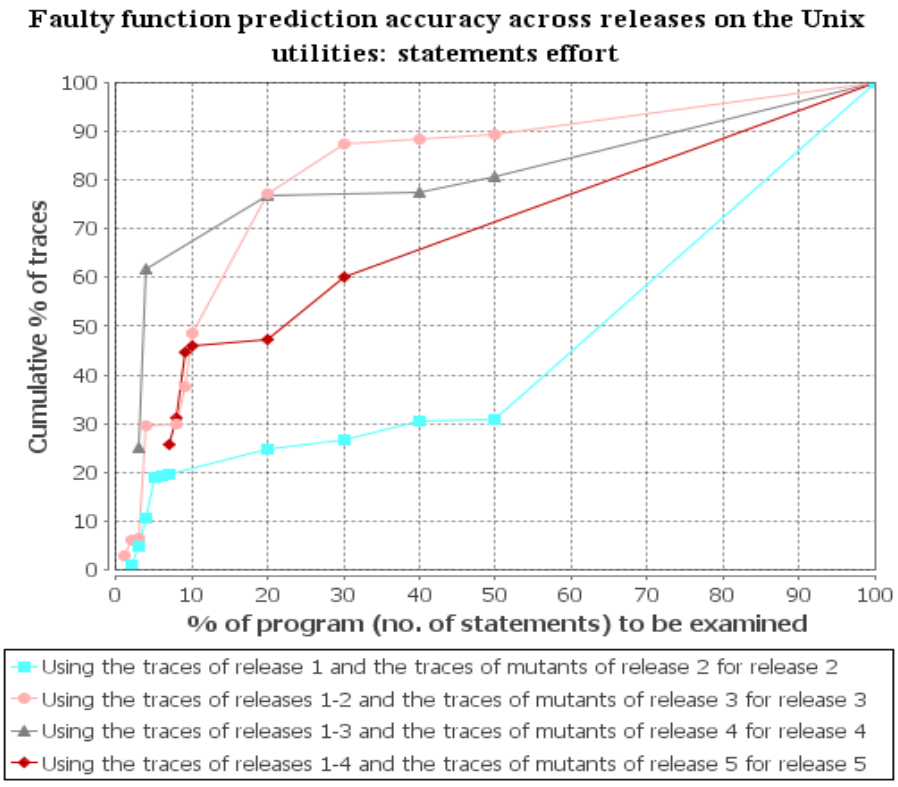


Figure 36: Faulty function prediction accuracy in terms of statements-effort on the actual failed traces of the current release using the failed traces of selected mutants and actual failed traces of the preceding release.

We can observe from Figure 34 to Figure 37 that effort in terms of statements mostly remained proportional to the effort in functions. This implies that in commercial or professional programs, functions are not distributed in extremely large to extremely small sizes: functions sizes are mostly proportional to each other.

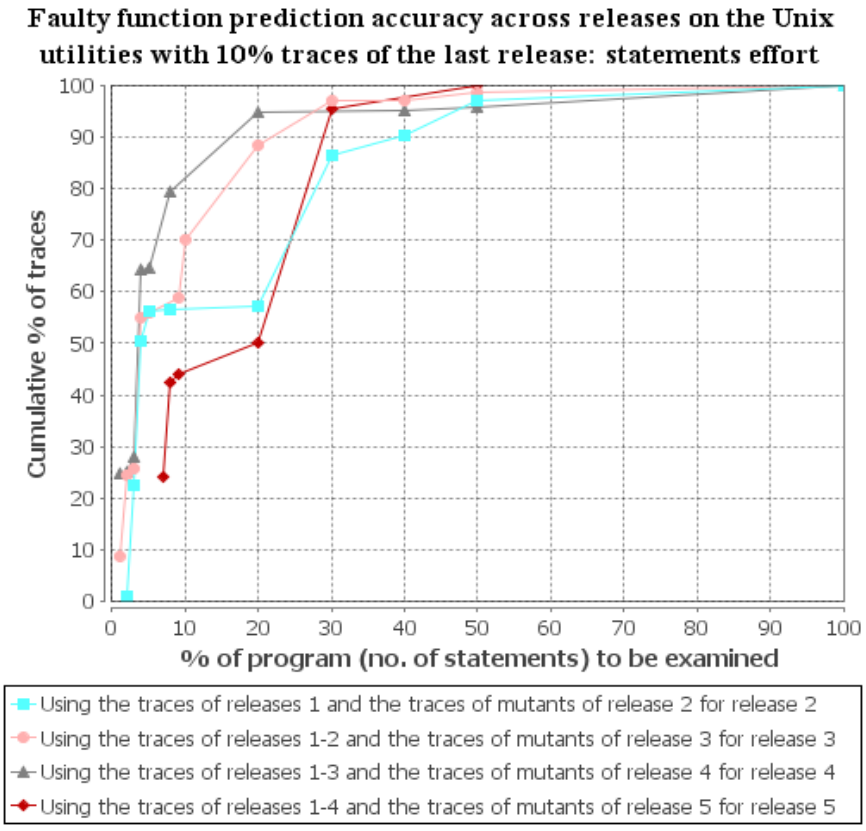


Figure 37: Faulty function prediction accuracy in terms of statements-effort on the actual failed traces of the current release using the failed traces of selected mutants, actual failed traces of the preceding release and 10% of the current release.

In Figure 36 and Figure 37, the estimation of effort in terms of statements is a pessimistic approach as we have summed all the statements of a function. In reality, a developer using the context of a fault (e.g., inputs, error message) can skip a function or may jump to another function after reviewing a few statements or no statements at all. A developer would not likely review *all* the statements of a function to determine whether that function is faulty or not. Therefore, in reality, the effort in statements would be better, and the graphs in Figure 36 and Figure 37 actually show the worst case. Further, a

developer can also use the previous faulty statements in a function as a starting point to investigate statements in a function. If 50-90% of the field failures are rediscoveries of the previous faults (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) then most of the previous faulty statements would likely be faulty again and this would further reduce the effort in statements.

3.8.3 Identifying Multiple Faulty Functions

In F007-plus, the multiple faulty functions are identified in the following two ways:

a) In Figure 26, we have shown that a trace is assigned a label of one faulty function. In the case of multiple faulty functions, a trace is assigned the label of multiple faulty functions and the decision tree is then trained on such traces. The decision tree identifies multiple faulty functions in a similar fashion to single faulty function as discussed in Section 3.4.3.

b) Previous method is better suited for the actual traces, when we already know multiple faulty functions for a trace, because artificially generating mutants for multiple functions can result into mutants for approximately 2n combinations of faulty functions (where n= total functions in a program). Therefore, in the case of mutants, F007-plus trains the decision tree on the traces of single faulty functions (as discussed in discussed in Section 3.4.3). F007-plus then predicts single faulty function for a multiple faulty function trace. The intuition is that if one of the faulty functions out of few faulty functions of the actual trace is predicted correctly then we consider that faulty function has been discovered for that trace. This is because it is likely that if a programmer discovers (or fix) one faulty function then other faulty functions occurring due to the same fault would get diagnosed (or fixed) automatically. Moreover, after fixing one faulty function if the failure appears again then the process can be repeated to discover other faulty functions.

Table 15: List of multiple faulty functions in release 3 of the Flex and the Grep program.

Flex	
Group 1	flexinit; yy_flex_realloc; myesc
Group 2	list_character_set; dump_associated_rules ; flexinit
Group 3	dump_associated_rules; yy_delete_buffer; set_input_file
Group 4	action_define; gen_NUL_trans
Group 5	myesc; yy_flex_realloc; global; yy_delete_buffer; set_input_file; reading; flexinit; gen_NUL_trans
Group 6	genctbl; global; gen_NUL_trans; dump_associated_rules; action_define; list_character_set
Grep	
Group 1	gcompile; nlscan; grepfile; page_alloc
Group 2	fillbuf; grepdir
Group 3	fillbuf; init_syntax_once; page_alloc'
Group 4	reset; lex
Group 5	grepfile; lex; init_syntax_once
Group 6	prtext; closure
Group 7	fillbuf; prline; lex

In order to demonstrate above two methods to identify multiple faulty functions, we enabled more than one fault simultaneously in the UNIX utilities by Do et al. (2005). The scripts provided by Do et al. (2005) enable faults one by one (one fault pertains to one faulty function) in the UNIX utilities, and run test cases on those enabled faults. This results into the failed traces of only one faulty function for each run of test suite. Similarly, we collected the failed traces of the mutants of single faulty functions one by one (as discussed in method 'b' above). Thus, our earlier results were based on single faulty functions.

We randomly selected the Flex and the Grep programs to illustrate the case of multiple faulty functions. We selected release three of these programs because release three resulted in the largest number of failed traces of single faulty functions—i.e., many test cases failed on release three of Flex and Grep. We randomly enabled multiple faults simultaneously for the release three of Flex and Grep: each time we enabled a different combination of faults. This resulted into several functions being faulty at the same time. The list of functions that we randomly made faulty for the Flex and Grep program is shown in Table 15. For example, for the Flex program, we enabled faults in six different groups of functions. In the first group, there are three faulty functions "flexinit", "yy_flex_realloc" and "myesc" that resulted by enabling different faults provided with

the Flex program (Do et al., 2005). We ran test suite on these groups of functions one by one, and collected all the failed traces; that is, in this case we relaxed the criteria of selecting only those faults which have 20% or lesser failed test cases. This also allowed us to evaluate F007 on the faults with more than 20% failed test cases.

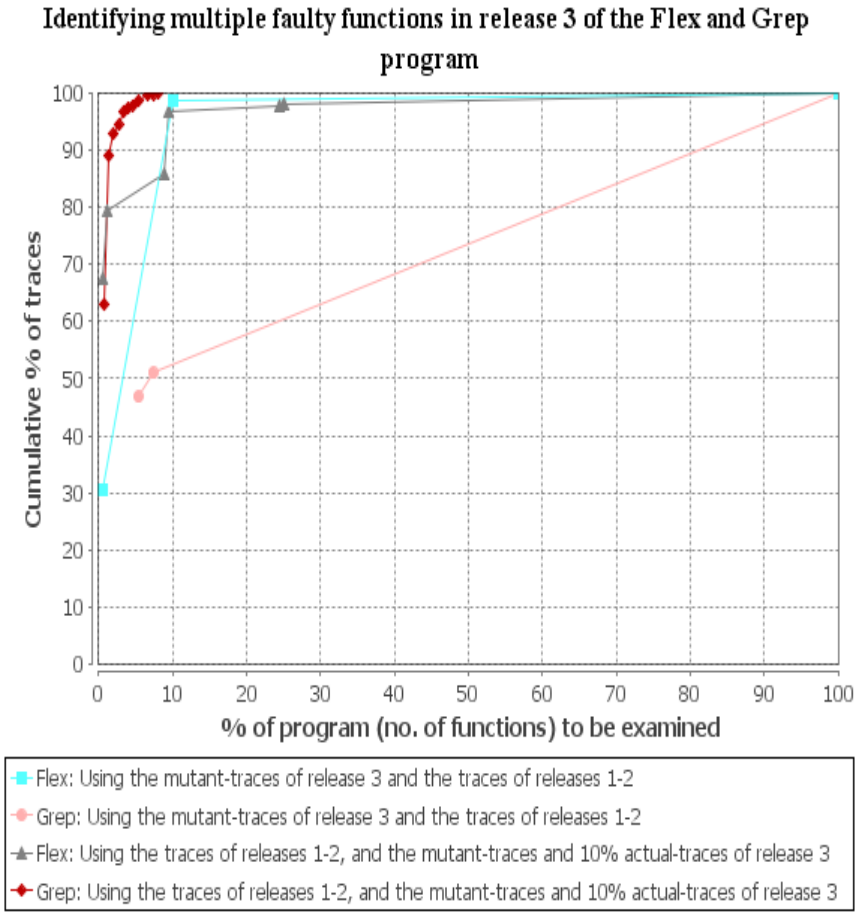


Figure 38: Identifying multiple faulty functions in the Flex and Grep program using mutant traces and actual traces.

In Figure 38, we show the results of identifying multiple faulty functions by using F007-plus. First we identified faulty functions in the traces of multiple-faulty-function-release-three using the traces of mutants of releases three and the actual failed traces of release one and two. If any one of the faulty function predicted by F007-plus matched one of the faulty functions of the trace, we considered that the faulty function was identified. For example, in the case of failed traces of group 1 of Flex program (Table 15), the first

faulty function that matched from the list of predicted faulty functions from F007-plus was “flexinit”, and “flexinit” was listed at the rank 1 of the predicted faulty function list.

In Figure 38, the accuracy of identifying faulty functions in multiple-faulty-function-release-three using the traces of mutants and failed traces of prior releases is: (a) 50% on the review of 7% of the program for the Grep; and (b) 98% on the review of 10% of the program of the Flex. This shows that even if there are multiple faults in different functions, one of the faulty functions can still be identified -- due to the similarity of function-calls between the multiple-faulty-function trace and the identified faulty functions trace. This again confirms the similarity of function-call traces among the faulty function's trace. Also note that it is not necessary that failed traces of multiple faults are triggered by all the multiple faults. It is possible that on running some test cases one particular fault in a faulty function is executed, hence resulting in the function-call traces similar to single faulty function.

In the case of Grep, Figure 38 shows that the accuracy is only 50% and in the rest of the cases faulty functions were not discovered at all. One of the reasons for no discovery of faulty functions in the remaining cases is that the training set did not contain failed traces of some faulty functions or contained very few. For example, in the case of “group 1” of Grep program (i.e., gcompile, nlscan, grepfile, and page_alloc) only one trace of the function “gcompile” and four traces of the function “grepfile” were present in the training set. This means that the decision tree had not enough knowledge and it didn’t predict faulty functions.

Finally, Figure 38 also shows the results of F007-plus by using mutant traces, failed traces of previous releases, and the 10% failed traces of multiple-faulty-function-release. In other words, the training set also includes the failed traces of actual multiple faulty functions (i.e., traces with the label of multiple functions as described in method ‘a’). Figure 38 shows that faulty functions in approximately 80% of the failed traces can be identified by reviewing 1% of the program for both the Flex and the Grep, when 10% failed traces of the multiple-faulty-function-release were also used. The inclusion of actual 10% traces overcomes the limitation of not discovering the faulty function when

previous traces do not have that faulty function. This is significant (as mentioned before) when 50-90% are rediscoveries of the same fault (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003).

Thus, the results in this section show that F007-plus can effectively identify faulty functions in the failed traces of multiple faulty functions. The results in this section also show that mutants can accurately identify actual faults with high accuracy. This improves the answer to research question (Q2).

3.8.4 Rules of Decision Tree in Understanding Fault Proneness of Faulty Functions

Decision tree model actually generates rules from the independent variables in a training dataset and use those rules to predict a dependant variable (see Section 3.4.3). In this section, we show examples of decision trees (i.e., rules) for randomly selected functions. These rules are useful in understanding why a particular function could be faulty and which unique execution paths mostly lead to that faulty function. For example, if a particular function is found faulty in a large number of traces and due to different faults, then a programmer can analyze these rules to find out which execution paths are causing that function to be faulty and perform extensive testing on the functions of those paths.

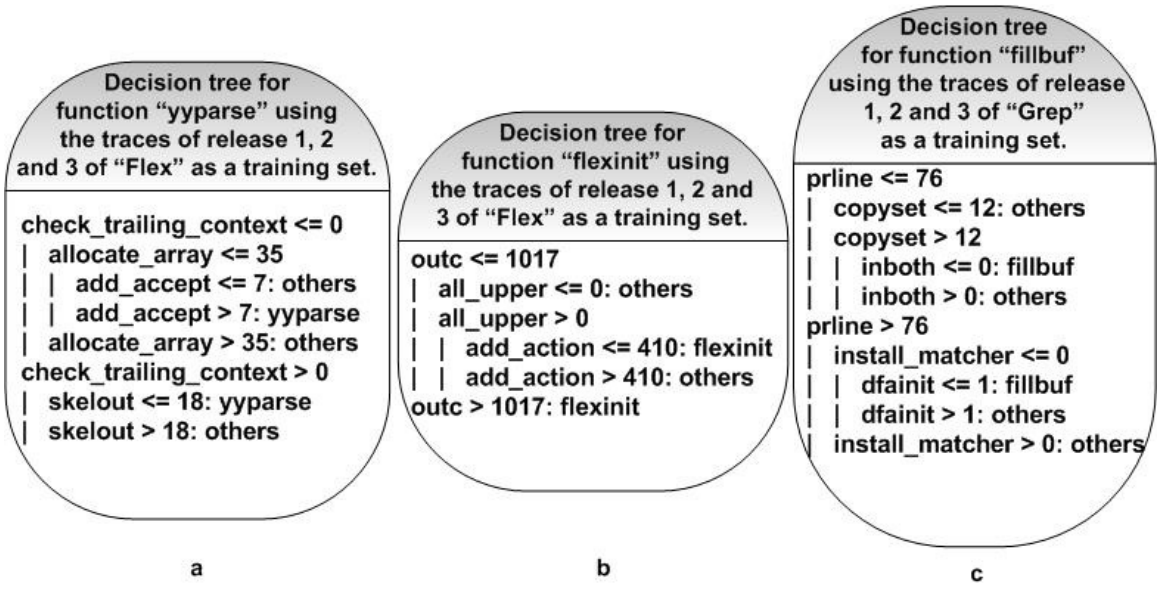


Figure 39: Decision tree models for faulty functions of Flex and grep program.

In Figure 14 (‘a’, ‘b’, and ‘c’), we show the three decision tree models, generated using the one-against-all approach, for functions of the Flex and Grep program. For example, in Figure 14 part ‘a’, the rules are read as if occurrence of the function “check_trailing_context” is less than equal to zero, and occurrence of “allocate_array” is less than equal to 35, and “add_accept” is greater than 7, then the faulty function is “yyparse”. In a similar manner, other rules in part ‘a’, ‘b’ and ‘c’ of Figure 14 can be analyzed. Thus, these rules actually provide an abstraction of a collection of faulty traces belonging to faulty functions, and provide a succinct human-readable view of suspicious function-calls (out of many function-calls in traces) leading to a faulty function.

3.9 Summary of the Findings

In this section, we summarize all the results of this study:

- Different faults in the same function do not occur with the exactly the same occurrences of function-calls, but different faults in the same function do have similar or closely related function-call occurrences (see Section 3.7.2).

- A conclusion that: “A group ‘M_i’ of related functions have similar function-call traces when a fault occurs in the functions of that group ‘M_i’; but the function-call traces of ‘M_i’ 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 $M_i \subset N$ and $M_k \subset N$ and $N=\{functions \mid functions \in program\}$.”(See Section 3.7.1.) This answers the first research question (Q1).

- Faulty functions in 50-60% of the actual traces can be identified on reviewing 20-30% of the code by making every function artificially faulty (i.e., using mutants) and using the traces of all those mutated functions (see Figure 30 and Figure 31 in Section 3.7.1). This answers the research question (Q2); but the accuracy of identifying faulty functions remains low when using the mutant traces of all the faulty functions.

- A new strategy F007-plus that improves F007-basic-- that identifies only rediscovered faulty functions by using prior faults--by identifying new and old faulty functions using mutants and prior faults (see Section 3.5 and Section 3.8).

- When using F007-plus, faulty functions in approximately 30-80% of the failed traces in the current release can be identified using the failed traces of the mutants of the suspected functions of the current release and failed traces of the previous releases (see Section 3.5).This improves the answer to research question (Q2). If 10% traces of the current release are used then faulty functions in 77-97% of the failed traces can be identified by reviewing approximately 3% of the program (see Section 3.8)

- The use of 25-30 mutant traces per function is sufficient to discover faulty functions in the actual traces (see Section 3.7).

- Results show that developer's effort in terms of the review of statements mostly remains proportional to the developer's effort in terms of the review of functions (see Section 3.8.2).

3.10 Comparison Against Other Techniques

In Section 3.2, we described the characteristics of prior techniques and how they differ from our technique. In this section, we shall further articulate the significant differences with the closely related techniques, which will get clearer now.

In Figure 40, we show the results of F007-basic (Murtaza et al., 2010) by training the decision tree on the actual failed traces of earlier releases and identifying faulty functions in the following releases of the UNIX utilities—that is, without using mutants. Figure 40 shows the accuracy of identification of faulty functions on all the four UNIX utilities (i.e., Flex, Grep, Gzip and Sed) and the results were obtained in the same manner as described in Section 3.8.

For example, in Figure 40, the series "using release 1 for release 2" shows that by training the decision tree on the failed traces of release 1 (training-release) of the four UNIX utilities, approximately 10% of the faulty functions can be identified in release 2 (test-release). This identification requires only the review of 7% or less of the code (functions) for the four UNIX utilities. Similarly, Figure 40 shows the result on other releases of the UNIX utilities by using the traces of all the preceding releases as the training set and the following releases as the test set.

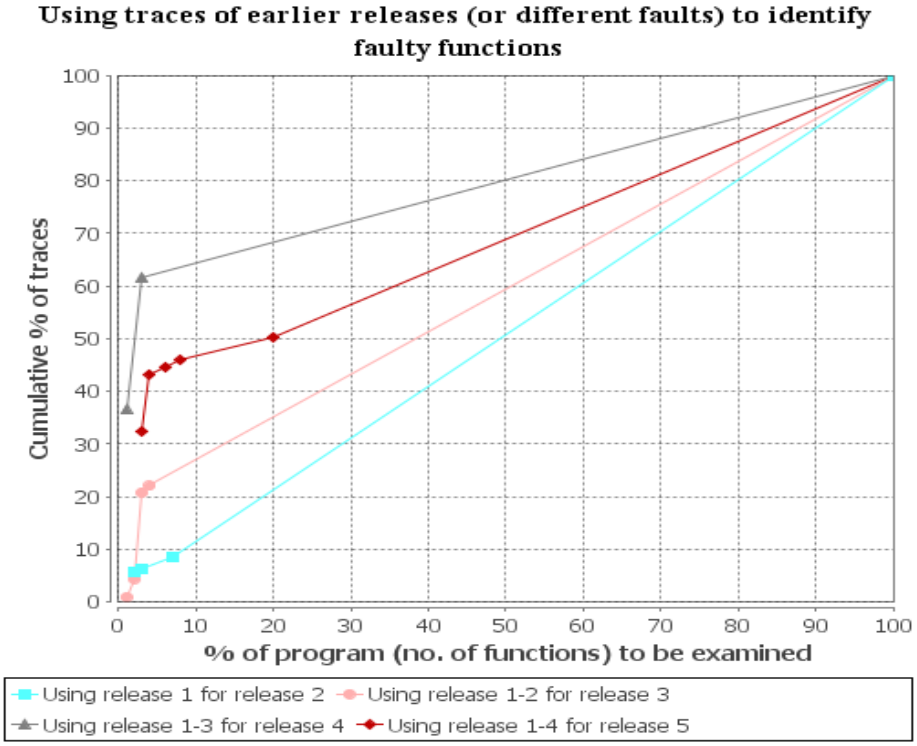


Figure 40: F007-basic on the UNIX utilities.

A limitation of the F007-basic approach is that faulty functions can only be identified in the current release if they are present in the previous release—new faulty functions can not be identified. This problem can be mitigated by using the mutants of the expected faulty functions of the current release with the traces of previous release—i.e., using F007-plus. We show the results of F007-plus on the UNIX utilities in Figure 41 which is the same as Figure 34. Results of F007-plus, in Figure 41, are annotated as R2, R3, R4, and R5 for release 2, release 3, release 4 and release 5 respectively. It can be observed that F007-plus (by using mutants) improves F007-basic by 10-60% on reviewing 20% of the code. F007-plus can identify 30-80% of the faulty functions in the traces of the succeeding release of the UNIX utilities by using the failed traces of prior releases and mutants of the current release. F007-plus, however, needs to be improved further to identify the majority of the faulty functions within the review of 10% of the code: that is, more research in the area of identification of the suspected faulty functions can improve F007-plus further.

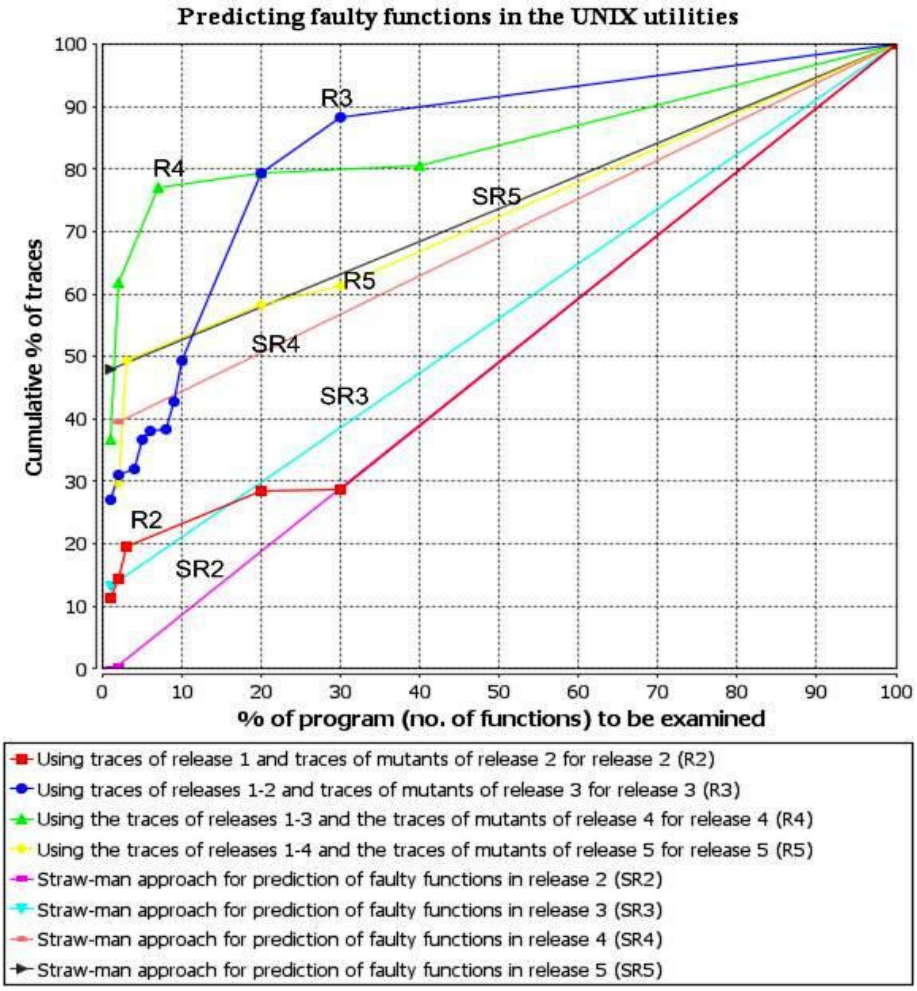


Figure 41: F007-plus on the UNIX utilities and straw-man approach for prediction of faulty functions.

Figure 41 also compares F007-plus with a straw-man approach. In the staw-man approach, we considered that for every new failed trace in a current release, a developer predicts the function with the largest LOC and the function found faulty in the largest number of failed traces of previous releases. The results are shown in Figure 41 by annotations SR2, SR3, SR4 and SR5 for release 2, 3, 4 and 5 respectively. For example, for release 2 (SR2), less than 1% of the failed traces were correctly resolved by reviewing 2% of the code. For the remaining 99% of the traces, 100% of the code was required to be reviewed. (Recall that straight line, when there are no points on a series in a graph, means that a developer needs to predict faulty functions by himself or herself.) The

difference between the accuracy of prediction of faulty functions seems to be low between F007-plus and the straw-man approach in the case of release 2 and release 5. In the case of F007-plus, the accuracy of prediction is low when test cases did not fail on mutants and there are not enough traces belonging to particular faulty functions in the training set. In the case of straw-man approach, accuracy of prediction is actually very low except in release 5, and the accuracy of predictionof the straw-man approach overlaps with F007-plus in Figure 41 only when a developer is predicting faulty functions randomly (i.e., at a straight line). Thus, F007-plus can effecticiently identify faulty functions across releases. Moreover, if only a small percentage (e.g., 10%) of failed traces of a current release is available then 70-90% of the failed traces can be identified by reviewing less than 10% of the program, as shown in Figure 35.

Other fault discovery techniques such as recording function sequences (Dallmeier et al., 2005; Di Fatta et al., 2006) to identify faulty functions (Di Fatta et al., 2006) and faulty classes (Dallmeier et al., 2005); evaluating statement coverage to identify faulty statements (Jones and Harrold, 2005; Wong et al., 2007); and statistical debugging (Chilimbi et al., 2009; Liu et al., 2005; Zheng et al., 2004) have the following major differences with our technique: (a) our technique can discover faults in a single trace using the previous traces (e.g., from mutants); whereas all these techniques identify a fault in a collection of passing traces and failing traces related to the same fault (version) [29]; (b) our technique incurs less overhead on deployed system as it does not require many passing traces and failing traces to be collected for a given fault; and (c) most of these techniques are suitable for in-house testing where passing and failing traces are readily available, whereas our technique is suitable for the field traces where only a few failed traces are available to find a fault. Thus, these techniques are related to our technique but they are not directly comparable and suited for a different purpose.

[29] A version is equivalent to one fault in the Space program or the UNIX utilities. These third party techniques identify fault in a collection of passing and failing traces of one version; whereas our technique identify faults in every failed trace of a version using prior failing traces.

Another, closely related technique focusing on field failures, proposed by Podgurski et al. (2003), clustered together failed (function level) execution profiles according to the same faulty source file. They evaluated their technique on programs such as GCC, Javac and Jikes, and it resulted in 57%, 30% and 29% clusters (groups of failed traces for a file) with the same source files, respectively. The majority of clusters contained more than one file making it difficult for manual investigation, specially the fine-grained code. Our technique discovers faults in failed traces at a finer-grain level— function, and with an accuracy of: (a) 25-80% on reviewing 10% of the code (functions) when using mutant traces and traces of prior releases (see Figure 34); and (b) 70-95% on reviewing 10% of the code (functions) when using only 10% failed traces of the current release (see Figure 35).The differences of this paper with other related techniques are already described in Section 3.2.

In Figure 23, we showed an example of the function-call level execution trace having both "function entry" *and* "function exit" events. However, we have found, in our earlier experiments (Murtaza, et al., 2010), that either "function entry" or "function exit" events, by themselves, are adequate to predict the fault origin (Murtaza, et al., 2010). Their combined use does not improve the success rate. This finding also implies that the overhead on the deployed systems will be half of a normal (function-call) failure trace if only "function entry" or "function exits" is collected for software—reducing the size of a trace to half as well (Murtaza, et al., 2010). Thus, in all the results we have shown in this paper, we have only used "function exit" events.

3.11 Threats to Validity

In this section, we describe certain threats to the validity of the research results obtained through our employed research process. We classify threats into four groups: conclusion validity, internal validity, construct validity, and external validity.

3.11.1 Conclusion Validity

Conclusion validity is concerned with our ability to draw the correct conclusion about the relations between treatment and outcome of an experiment (Wohlin et al., 2000).

A threat to conclusion validity belongs to random variations in mutant traces. We randomly chose 3 mutants per function, but on some programs test cases did not fail on the mutants (see Figure 34 for release 2). It is possible that different selected mutants for the same function might result in failing test cases and variations in accuracy for some programs. For example, in Figure 33, we have observed that by using different random mutant traces per function, the accuracy of identification of faulty function is about 75% on the review of 10% of the program; whereas in Figure 33 we showed 70% accuracy on using other mutant traces. This may be due to the function-call path that a fault generated using mutant has taken in the traces. However, this threat is mitigated by the fact that we use 25-30 mutant traces per faulty function, repeated experiments on about 19 real world programs (i.e., 4-5 releases of the four UNIX utilities, and one release of the Space program; see Table 11), and we collected different faulty paths for a function by selecting three different mutants per function. Also, as we have found out in the experiments that function-call traces of related functions are similar, therefore the results are not going to be different from the ones shown in this paper.

3.11.2 Internal Validity

Internal validity is concerned whether the relationship between treatment and outcome is causal, and not due to any confounding factors (Wohlin et al., 2000).

A threat to internal validity can exist in the implementation of different algorithms because an incorrect implementation can influence the output. For example, we wrote shell scripts to automate mutant trace collection, developed a Java program to automatically extract functions and their locations from C programs, modified the mutant tool to randomly generate three mutants per function, and developed F007-plus in Java and MySQL. In our investigation, this threat is mitigated by making our implementation reliable, for example, by manually investigating the outputs (e.g., we manually verified for some programs that random mutants were correctly generated for a function).

3.11.3 Construct Validity

Construct validity refers to the extent to which the experimental settings actually reflect the construct under study (Wohlin et al., 2000).

A threat to construct validity is related to the use of code metrics. We used four code metrics (see Section 3.8) for a function as independent variables to predict faulty functions for the future releases. A different set of code metrics can be used if they result in better accuracy. For example, a factor for the accuracy for release 2 in Figure 34 is low, apart from no failing test cases, is that some of the suspected functions using code metrics were not predicted. Also the cost ratios in Table 12 for the "Sed" program are quite high, which could mean that a different set of code metrics can be used which may result in a lower cost ratio for the "Sed" program". However, the focus of this research is not to determine which set of code metrics are suitable, but to demonstrate that we can leverage the code metrics based techniques (Basili et al., 1997;Emam et al., 2001;Lounis and Ait-Mehedine, 2004); to generate mutants to identify faulty functions in the actual traces. Further, we have also described the criteria for selecting the cost ratio using training set, which can be used to identify majority of "faulty" functions in the future releases. Also, the results in Section 3.8 on other releases show that this method has a potential to generate right set of mutants to identify faulty functions accurately, if the test cases fail, in the actual failed traces.

A threat to construct validity exists in the use of failed field traces for fault discovery by F007-plus. Consider, automated failure reporting such as in Mozilla, Net Beans, and Visual Studio. This failure reporting facilitates fault localization by providing contextual information, traces, etc. to the developers. It may be possible that such large number of traces may contain passing traces. In such cases, pass-fail classification techniques (Bowring et al., 2004; Haran et al., 2007) or a technique to collect only function-calls related to the fault (Elbaum et al., 2007) (which are complementary to our work) can be used to classify a trace as passing or failing. However, if a trace is captured at the time of a fault, then F007-plus will identify faulty functions in that trace. This is because if a trace is captured at the time of a fault then it would encompass the sequences of function-calls contributing to faulty functions.

3.11.4 External Validity

External validity refers to the ability to generalize results of an experiment to industrial practice (Wohlin et al., 2000).

Second threat to validity is that we have experimented only on the medium sized commercial programs. This technique is still to be validated on the very large industrial scale software application, and caution is advised in unproven context. However, many of the significant papers in the field of fault localization have used the Space program (Andrews et al., 2005; Jones and Harrold, 2005; Wong and Qi, 2006) and the UNIX utilities (Zhang et al., 2009). This shows the significance of our results.

Another threat to external validity exists in that a new failed trace (e.g., from a new release) could contain a function-call that does not exist in the training set. In order to build the decision tree, we need to add those newer functions in the training set of the failed traces of previous releases with the confidence of '0', or discard the new function-calls and use the ones already in the trained tree. In the case of F007-plus, we have mitigated this threat by using the mutant traces of the last release, which facilitated in including the functions introduced in the new release. Also, before generating the decision tree from the traces of the training set, we checked if there were new functions in the traces of the test set. If there were new functions, we added the names of those functions with the value of '0' in the training set.

3.12 Conclusions

Identification of the origin of a fault remains an arduous and time consuming activity of corrective maintenance, which can consume approximately 30-40% of corrective maintenance time (Proprietary Workshop, 2008). A number of techniques proposed for deployed software focus on: the classification of field profiles into failed or successful executions (Bowring et al., 2004; Haran et al., 2007); clustering field profiles (Liu and Han, 2006; Podgurski et al., 2003); rediscovery of crashing faults (Brodie et al., 2005; Lee and Iyer, 2000); statistical debugging (Chilimbi et al., 2009; Liu and Han, 2006); and rediscovery of crashing and non-crashing faults (Murtaza, et al., 2010).

This paper discusses two new questions that have not been dealt with in the literature before: (Q1) Are the function-call level traces of some faulty functions similar and that of some other faulty functions different? (Q2) Can the faults generated using mutants (artificial faults) be used to discover the actual faults? These questions are important

because we can reduce the time and effort spent in the corrective maintenance by using artificial faults (i.e., mutants) to discover the actual faults. We investigated the answers to these research questions by experimenting on one release of the Space program, and four to five releases of the Flex, Grep, Gzip and Sed programs (Do et al., 2005).

This paper contributes by identifying: "A group 'M_i' of related functions have similar function-call traces when a fault occurs in the functions of that group 'M_i'; but the function-call traces of 'M_i' 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 $M_i \subset N$ and $M_k \subset N$ and $N=\{functions \mid functions \in program\}$". This answers the first research question (Q1).

This paper also contributes by proposing a new strategy F007-plus that improves our prior work F007-basic (Murtaza, et al., 2010) –that identifies only rediscovered faulty functions--by identifying new and old faulty functions using mutants and prior faults (see Section 3.8). F007-plus trains the decision trees on the traces of the mutants of suspected faulty functions in a current release and the failed traces of actual faulty functions to identify faulty functions in actual failed traces (see Section 3.5): the suspected faulty functions were identified using the cost sensitive learning strategy on the code metrics of programs.

When using F007-plus faulty functions in approximately 30-80% of the failed traces in the following release can be identified using the failed traces of the mutants of the suspected functions of the current release and failed traces of the previous releases (see Section 3.8.1). This answers research question (Q2). If 10% traces of the current release are used then faulty functions in 77-97% of the failed traces can be identified by reviewing approximately 3% of the program (see Section 3.8.1). If F007-plus is compared with F007-basic (which only uses traces of previous releases) then F007-plus improves F007-basic by 10-60% on the review of 20% of the code (see Section 3.10).

Our technique is thus invaluable for deployed software when it is not feasible to collect many (passing and failing) traces. This is mostly the case with deployed software due to overhead incurred and time spent in collecting traces.

We have experimented only on the medium size commercial programs and this technique still requires validation on very large scale industrial programs. Also, we have used code metrics based classification to identify the possible faulty functions and mutants (see Section 3.8) in a future release. One of the future research issues is to explore this further to exactly identify the right set of expected faulty functions to generate the exactly relevant mutants for training. This will improve the accuracy of identification of the faulty functions in the actual traces, and will result in the reduced effort of mutants and the failed traces generation.

3.13 References

Agrawal, H.; Horgan, J.R.; London, S.; Wong,W.E.; "Fault Localization using Execution Slices and Dataflow Tests", *Proc. of Int'l Soft. Symp. on Reliability Eng*., IEEE, France, Oct., 1995, pp.143-151.

Andrews, J. H.; Briand, L. C.; and Labiche, Y, "Is mutation an appropriate tool for testing experiments?", *Proc. of the 27th Intl. Conf. on Sof. Engg.,*ACM, St. Louis, USA, May, 2005, pp. 402-411.

Andrews, J.H.; Briand, L.C.; Labiche, Y.; Namin, A.S.; , "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Transactions on Softw. Eng.,* Vol.32, No.8, Aug. 2006, pp.608-624.

Basili, V. R., Condon, S. E., El Emam, K., Hendrick, R. B., and Melo, W., "Characterizing and modeling the cost of rework in a library of reusable software components," *Proc. Conf. on Software Engineering*, ICSE, ACM , Boston, USA, May , 1997, pp. 282-291.

Bowring J.F.; Rehg J.M.; and Harrold. M.J; "Active Learning for Automatic Classification of Software Behavior", *SIGSOFT Soft Eng. Notes,* Vol. 29., No. 4, ACM, USA, July 2004, pp. 195-204.

Brodie, M.; Sheng Ma; Lohman, G.; Mignet, L.; Modani, N.; Wilding, M.; Champlin, J.; Sohn, P., "Quickly Finding Known Software Problems via Automated Symptom Matching", *Proc. of Second Int'l Conf. on Autonomic Computing, ICAC 2005*, IEEE CS, June 2005, pp. 101-110.

Chen M.; Accardi A.; Kiciman E.; Fox A.; Patterson D.; and Brewer E.; "Path-based Failure and Evolution Management", *Proc. of Int'l Symp. on Networked Systems Design and Implementation*, USENIX Association, CA,USA, March 2004,pp. 309-322.

Chilimbi, T. M.; Liblit, B.; Mehra, K.; Nori, A. V.; Vaswani, K; "HOLMES: Effective Statistical Debugging via Efficient Path Profiling.", *Proc. of 31st Intl. Conf. on Soft. Eng*., IEEE CS, Canada, May, 2009, pp.34-44.

Dallmeier, V.; Lindig, C.; Zeller, A, "Lightweight Defect Localization for Java", *ECOOP 05- Object Oriented programming*, Lecture Notes in Computer Science, Springer, Glasgow, UK, August 2005,pp 528-550.

Di Fatta, G.; Leue, S.; Stegantova, E; "Discriminative Pattern Mining in Software Fault Detection", *In Proc. of 3rd Int'l Workshop on Soft. Quality Assurance*, ACM, Oregon, USA, Nov. 2006, pp. 62-69.

Ding, X.; Huang, H.; Ruan, Y.; Shaikh, A.; and Zhang, X.; "Automatic Software Fault Diagnosis by Exploiting Application Signatures". *Proc. 22nd Conf. on Large Installation System Admin.*, USENIX Ascoiation, *San Diego,* CA, USA, Nov., 2008, pp. 23-39

Do H. and Rothermel. G., "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques". *IEEE Transactions on Soft. Engg.*, Vol. 32, No. 9, IEEE, USA, Sep. 2006, pp. 733–752.

Do, H.; Elbaum, S. G.; and Rothermel, G.; "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Journal of Empirical Soft. Eng.*, vol. 10, Kluwer Academic Publisher, USA, Oct. 2005, pp. 405-435

Emam E.,K; Melo,W.; Machado, J.,C.; "The prediction of faulty classes using object-oriented design metrics", *Journal of Systems and Software,* Vol. 56, No. 1, Elsevier Science Inc., USA, Feb. 2001, pp. 63-75.

Elbaum, S., Kanduri, S., and Andrews, A. "Trace anomalies as precursors of field failures: an empirical study". *Journal of Empirical Soft. Engg.*, Vol.12, No.5, Kluwer Academic Publisher,USA, Oct. 2007, pp. 447-469.

Etrace(Runtime Tracing Tool):http://ndevilla.free.fr/etrace/;March, 2008.

Gittens M., Kim Y., and Godwin D., "The vital few versus the trivial many: Examining the pareto principle for software", *Proc. of 29th Int'l Computer Software and Applications Conf. (COMPSAC'05)*, IEEE CS, Edinburgh, Scotland, July 2005, pp. 179-185.

Hao, D.; Pan, Y.; Zhang, L.; Zhao, W.; Mei, H.; and Sun, "A Similarity-aware Approach to Testing Based Fault Localization", P*roc. of 20th IEEE/ACM Intl. Conf. on Automated Soft. Engg.,*ACM, CA, USA, Nov. 2005, pp. 291-294.

Haran, M.; Karr, A.; Last, M.; Orso, A.; Porter, A.A.; Sanil, A.; Fouche, S.; " Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks", *IEEE Trans. on Soft. Eng.* Vol. 33, No.5, IEEE, USA, May, 2007, pp. 287-304.

Hutchins, M.; Foster, H.; Goradia, T.; Ostrand, T., "Experiments on The Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria," *Proc. of 16th Int'l Conf. on Soft. Eng.*, IEEE, Sorrento, Italy, May,1994 ,pp.191-200.

Jones, J. A. and Harrold, M. J., "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. of 20th Int'l Conf. on Automated Soft Eng.*, ACM, CA, USA,2005, pp.273-282.

Lee M. G.; Jefferson T. L.; "An Empirical Study of Software Maintenance of a Web-based Java Application", *Proc. of Int'l Conf. on Soft. Maint. (ICSM)*, IEEE CS, Budapest, Hungary, Sep.,2005, pp. 571-576.

Lounis, H.; Ait-Mehedine, L. "Machine Learning Techniques for Software Product Quality Assessment," *Proc. Conf. on Quality Software (QSIC)*, IEEE CS, Germany, Sep, 2004, pp. 102-109

Lee, I.; Iyer, R., "Diagnosing Rediscovered Problems Using Symptoms", *IEEE Transactions on Software Engineering*, vol. 26, no. 2,IEEE USA, Feb. 2000, pp. 113-127.

Liu, C. and Han, J., "Failure proximity: a fault localization-based approach". *Proc. of the 14th SIGSOFT Sym. on Foundations of Software Engineering*, ACM, Portland, USA, Nov. 2006, pp. 45-56.

Liu, C.; Yan, X.; Fei, L.; Han, J.; Midkiff, S. P.; "SOBER: Statistical Model-Based Bug Localization", *SIGSOFT Softw. Eng. Notes*, Vol 30, No.5, ACM, USA, Sep,2005, pp. 286-295.

Mayer J. and Schneckenburger. C., "An Empirical Analysis and Comparison of Random Testing Techniques".*Proc. of Intl. Symp. on Empirical Soft. Engg.*, ACM, Brazil, Sep., 2006, pp. 105–114.

Murtaza, S.,S.; Gittens, M.; Li, Z., Madhavji, N.H.; "F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field*", Proc. 2010 conference of the centre for advanced studies on collaborative research: meeting of minds (CASCON 2010)*, ACM, Ontario, Canada, Oct. 2010, pp. 57-71.

Offutt, A., J. and Untch, R., H., "Mutation 2000: uniting the orthogonal", in *Mutation Testing for the New Century*, Kluwer Academic Publishers, Wong W,E., (Ed.), USA, 2001, pp. 34-44.

Ostrand T. J., Weyuker E., and Bell R. M., "Predicting the location and number of faults in large software systems", *IEEE Trans. on Software Eng.*, Vol. 31, No. 4, 2005, IEEE, USA,pp. 340-355.

Podgurski, A.; Leon, D.; Francis, P.; Masri,W.;Minch, M.; Sun, J.; Wang, B, "Automated Support for Classifying Software Failure Reports", *Proc. Intl. Conf. on Software Eng.* ,IEEE CS, Portland, US, May,2003, pp. 465-475.

Polat, K.; Güneş, S, "A Novel Hybrid Intelligent Method Based on C4.5 Decision Tree Classifier and One-Against-All Approach for Multi-Class Classification Problems", *Journal of Expert Syst. Appl.* Vol. 36, No.2 , Pergamon Press, USA, Mar., 2009, pp. 1587-1592.

Proprietary workshop on a large commercial software, Sep., 2008.

Quinlan, J. R.; *C4.5: Programs for Machine Learning,* Morgan Kaufmann Publishers Inc., 1993.

Schach S. R.; Jin B.; Yu L.; Heller G. Z.; and Offutt J. "Determining the Distribution of Maintenance Categories: Survey versus Measurement", *Journal of Empirical Soft. Engg*. Vol. 8, No. 4, Springer, Netherlands, Dec., 2003, pp. 351-365.

So, S. S., Cha, S. D., and Kwon, Y. R., "Empirical evaluation of a fuzzy logic-based software quality prediction model," *Fuzzy Sets and Syst.*, Vol. 12, No. 2, Elsevier Science,The Netherlands, April, 2002, pp.199-208.

Ting, K.M.; "An instance-weighting method to induce cost-sensitive trees," *IEEE Transactions on Knowledge and Data Engineering, V*ol.14, No.3, IEEE, USA, Jun 2002, pp.659-665.

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

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; and Wesslén, A;. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, USA, 2000.

Wong, W. E. and Qi, Y., "Effective Program Debugging Based On Execution Slices and Inter-Block Data Dependency", *J. Syst. Softw.,* Vol.79 ,No. 7, Elservier Inc., USA, July, 2006, pp. 891-903.

Wong, W.E.; Yu Qi; Lei Zhao; Kai-Yuan Cai, "Effective Fault Localization using Code Coverage," *Proc. of 31st Int'l Conf. on Comp. Soft. & App.*,Vol.1, IEEE CS, China, July, 2007, pp.449-456.

Wood A., "Software reliability from the customer view", *Computer* , vol. 36, no. 8, IEEE CS, USA, August 2003, pp. 37-42

Yuan, C.; Lao, N.; Wen, J.; Li, J.; Zhang, Z.; Wang, Y.; and Ma, W; "Automated known problem diagnosis with event traces", *SIGOPS, OS. Syst. Rev.*, Vol. 40, No. 4, ACM,USA, Oct., 2006, pp. 375-388.

Zhang, Z.; Jiang B., and Wang, X., "Capturing Propagation of Infected Program States". Proc. *Intl. Conf.* on *Foundations of Soft Engg.*, ACM, Netherlands, 2009, pp. 43-52.

Zheng A.X.; Jordan M.I., Liblit, B.; Aiken, A, "Statistical Debugging of Sampled Programs", In Advances in Neural Information Processing Systems, MIT Press, Cambridge, MA, US,2004. pp. 9-18.

Chapter 4

4 Emerging Theory

4.1 Introduction

The main objective of this dissertation was to develop a technique to automatically identify fault origin in the traces of field failures. However, we observed that findings of this thesis during the course of this research could lead to the foundation of an emerging theory in the field of software fault localization. We, thus, propose an emerging descriptive theory for the relationships between function-call traces of different faults. This emerging theory is developed in a bottom-up fashion according to the results obtained from the empirical findings; that is, it directly follows from the observations (Sjøberg et al., 2008). The propositions of emerging theory are formed in Section 4.2 by *hypothetico-inductive* model (Sjoberg et al., 2008), and this emerging theory is stated in Section 4.3 using the propositions formed in Section 4.2. The propositions are evaluated using the criteria for measuring the goodness of a theory (Sjøberg et al., 2008) in Section 4.4. We explain the implications of this theory in Section 4.5 and conclude this chapter in Section 4.6.

4.2 An Emerging Theory

In this section, first we explain the background information on how theory is formed on the basis of literature. Sjøberg et al. (2008) 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). Sjoberg et al.'s (2008) method to develop theoretical propositions is also used by Ferrari (2010), in his Ph.D. thesis, to develop an emerging theory on the interaction of system architecting and requirement engineering. The work on an emerging theory in this thesis is inspired from his work (Ferrari, 2010).

Sjøberg et al. (2008) develop their framework for describing SE theories by using the other frameworks proposed for other sciences such as social sciences, behavioral sciences, business management, etc. For example, the above three levels of propositions are based on the work of social research (Merton, 1968; Yin, 1984) and information systems research (Caroll & Swatman, 2000). Sjøberg et al. (2008) argue that software engineering (SE) theories are different from social and behavioral sciences. This is because SE theories are more applied and dependent on time and place at the current stage of development (Sjøberg et al., 2008); whereas theories in the social and behavioral sciences are independent of time and place (Cohen, 1989). For example: change in education and skill of a software engineer over time may change the validity of a theory; and the context of lab or industry as a placeholder may affect the validity of a theory (Sjøberg et al., 2008). Thus, we adopted the framework for describing SE theories by Sjoberg et al. (2008) to propose an emerging theory in this thesis.

In Table 16, we hierarchically organize different level of propositions emerging from this thesis. Level 1 proposition is observed directly from the empirical results of the studies in this thesis. Level 2 propositions are higher level abstract representations of Level 1 propositions. Both Level 1 and Level 2 propositions are testable and tested in their source studies or abstracted from the source studies. 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 (Sjøberg et al., 2008).

Table 16 first labels two studies conducted in this thesis as [S1] and [S2], and later it shows two theoretical propositions that arise from theses two empirical studies. Each proposition has two proposition levels according to Sjøberg et al. (2008) criteria and each level proposition references the source study from which it is observed. Level 1 propositions are observed from the study [S1] and [S2], whereas Level 2 proposition, which generalize Level 1 propositions, is abstracted from Level 1 propositions. Each level proposition is also assigned a unique number which will be used in explaining the propositions. Explanation of the propositions is as follows:

Table 16: Theoretical propositions arising from the empirical studies.

<table>
<tr><td colspan="2">[S1] (Chapter 2) F007: Finding Faulty Functions from the Function-call level Traces of the Field Failures.
[S2] (Chapter 3) Using Mutants to Discover New and Rediscovered Field Faults by Exploiting the Similarity of Traces among Different Faulty Functions.

P^L_I represents a unique proposition number, where L = level number and I=proposition id at that level.</td></tr>
<tr><td>Level 1 proposition</td><td>Level 2 proposition</td></tr>
<tr><td>(P^1_1) Faulty functions in 70-90% of failed traces can be identified on reviewing 20% or less of the program, when using the traces of at most one fault in 20% functions of the Space program and 20-100% functions of the Siemens suite [S1].</td><td rowspan="5">(P^2_1) A faulty function can be so identified if the traces of at least one fault in that function are already known; and the accuracy of identification increases with the decreasing proportion of faulty functions in the program.</td></tr>
<tr><td>(P^1_2) Faulty functions in 10-60% of failed traces in a succeeding software release can be discovered by reviewing 10% or less of the code, if traces of faults in less than10% functions of preceding software releases are known in the UNIX utilities and the large program (we used) [S1].</td></tr>
<tr><td>(P^1_3) Faulty functions in 30-80% of failed traces in a current release can be discovered on reviewing 20% of the program, when using failed traces of mutants (artificial faults) in approximately 10-40% functions of the current release and failed traces of faults in 10% functions of the preceding release of the UNIX utilities (we used) [S2].</td></tr>
<tr><td>(P^1_4) Faulty functions in 50-60% of the actual traces can be identified on reviewing 20-30% of the code by making every function artificially faulty (i.e., using mutants) and using the traces of all those mutated functions of the UNIX utilities and the Space program [S2].</td></tr>
<tr><td>(P^1_5) A group ‘Mi’ of related functions have similar function-call traces when a fault occurs in the functions of that group ‘M_i’; but the function-call traces of ‘M_i’ 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 $M_i \subset N$ and $M_k \subset N$ and $N=\{functions \mid functions \in program\}$ [S2].</td></tr>
</table>

4.2.1 Explanation of propositions (P^1_1 and P^1_2) from the study [S1]

During our experiments on the Siemens suite (Hutchins et al., 1994; Do et al., 2005) and the Space program (Do et al., 2005) in the study [S1], we have observed that if only a few traces (1-25%) of one fault in a function are known then F007 can identify the same

faulty functions with different faults in 70-90% of the failed traces on reviewing 3% and 20% of the program for the Space program and the Siemens suite, respectively (see Section 2.6.2.2). For the Siemens suite, 20% of the program-reivew was equivalent to 3 functions, and for the Space program 3% of the program was equivalent to 2-3 functions. 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.

Similarly, in Study [S1], we also found that by using the failed traces of faulty functions of previous releases, the same faulty functions in approximately 10-60% failed traces of future releases can be identified on reviewing approximately 5-10% of the code (see Section 2.6.2.2 and Section 2.7.6). For example, Section 2.6.2.2 (in Chapter 2) shows the results on different releases of the UNIX utilities and Section 2.7.6 (in Chapter 2) the results on the large program. In both Section 2.6.2.2 and Section 2.7.6, if faulty function was present in the previous releases then it was accurately identified (i.e., by reviewing less than 10% of the program). Moreover, when F007 was trained on more and more traces of earlier releases the accuracy of identification of faulty functions in the succeeding releases increased: because there were more common faulty functions across releases. This implies proposition P^1_2 and it also shows a similarity in function-call traces of differerent faults in the same functions.

In the study [S1], the Space program had about 20% faulty functions, and those functions were found faulty in actual development. Similarly, in the case of large commercial program, we found less than 10% functions faulty in our sample of traces, and the faults were actually found during the execution of software in the field. On the contrary, faults in the UNIX utilities and the Siemens suite were hand seeded. To keep the faults realistic in the UNIX utilities Do et al. (2005) seeded faults in those areas of code where changes were made by developers; this resulted into approximately 10% faulty functions in the UNX utitilies in our experiement. The Siemens suite, on the other hand, was a collection of seven small (non-commercial) programs with 20-100% faulty functions hand seeded by Hutchins et al. (1994), and, thus, the programs in the Siemens suite did not follow the 80-20 Pareto rule of software faults unlike other subject programs. This suggests that in

commerical programs a small proportion of functions are faulty, and using the traces of those faulty functions we can identify faulty functions in majority of the failed traces.

4.2.2 Explanation of propositions (P^1_3, P^1_4 and P^1_5) from the study [S2]

The proposition P^1_3 at Level 1 is observed from the study [S2], where we found that using the traces of mutants (i.e., artificially faults obtained by changing statements (Offutt et al., 2001)) in functions we can identify the same functions in the traces of actual faults. In the study [S2], we observed that we can estimate expected faulty functions in a current release using the code metrics of past releases (see Section 3.5), and we can accurately determine faulty functions in a failed trace by using the failed traces of those expected faulty functions and traces of faulty functions of prior releases. The expected faulty functions constituted approximately 10-40% of the total functions. For example, F007-plus in the study [S2] estimated: (a) 44-56 faulty functions in the Flex program; (b) 12-43 faulty functions in the Grep porgram; (c) 29-45 faulty functions in the Gzip program; and (d) 4-28 faulty functions in the Sed program. Moreover, less than 10% of the functions were actually faulty in the previous releases, some of which overlapped with the expected faulty functions. Thus using faults in approximately 10-40% of the functions, we were able to identify faulty functions in 30-80% of failed traces in Section 3.8.1. Again, in this case (Section 3.8.1), if the training set had traces of those faulty functions that were faulty in succeeding releases, then the accuracy of identifying faulty function was high (e.g., 80%); otherwise the accuracy was low (e.g., 30%) when training set was missing the traces of faulty functions. The high accuracy of identification of faulty functions, when traces of those faulty functions (with different mutant faults) were present in the training set, also show that there was a similarity in traces of different faults in functions.

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 [S2]. In the study [S2], we made every function in the Space program and the UNIX utilities artificially faulty using mutation (see Section 3.7.1). The results showed that faulty functions in 50-60% of the failed traces were discovered on reviewing 20-30% of the program, when every function was faulty (i.e., propostion P^1_4). These results imply that

function-calls of closely related functions overlap when faults occur in them; that is, traces of faulty functions form groups (see Section 3.7). In other words, we observed that different faults in a group of related functions occur with similar function-call traces and traces of faults in one group of functions differ from the traces of faults in other groups of functions. This results in proposition P^1_5. This propition P^1_5 also means that we can identify faulty functions with high accuracy (as in P^1_1, P^1_2, P^1_3) in failed traces if in a previous collection of failed traces (i.e., training-set) only a small percentage (i.e., up to 40%) of functions are faulty. If the training-set has faulty traces of every function then the accuracy of identification would be low. If 20% of the code is responsible for 80% of the faults (Boehm and Basili, 2001; Gittens et al., 2005; Ostrand et al., 2005) then a small proportion of functions will always be faulty in a program.

4.2.3 Explanation of proposition P^2_1

Finally, proposition P^2_1 generalizes from the propositions at level 1 by using the fact that: (a) a faulty function can be accurately identified from traces if traces of at least one (same or different) fault of that faulty function are present in the previous collection of traces (i.e., using proposition P^1_1, P^1_2, and P^1_3); and (b) accuracy of identification of faulty functions would be higher if a small proportion of functions are faulty (i.e., using proposition P^1_1, P^1_2, P^1_3, P^1_4, and P^1_5) because traces of closely related faulty functions overlap.

4.3 Emerging Theory Statement

Overall, based on the propositions at Level 1 and Level 2 this emerging theory can be stated as:

> *"A faulty function can be so identified if the traces of at least one fault in that function are already known; and the accuracy of identification increases with the decreasing proportion of faulty functions in the program."*

4.4 Evaluating Emerging Theory

Sjøberg et al. (2008) 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). Ferrari (2010) also used Sjøberg et al. (2008) criteria for the evaluation of goodness of an emerging theory on the interaction of system architecting and requirement engineering. We have also adopted the following criteria from the work of Sjøberg et al. (2008). In fact, Sjøberg et al. (2008) criteria are almost similar to the work of Boehm and Jain (2005) criteria and can be considered as their (Boehm and Jain, 2005) representative. Sjøberg et al. (2008) (and also Boehm and Jain (2005)) criteria were adapted for SE theory evaluation from other disciplines such as Business Management (Bacharach, 1989), Psychology (Haig, 2005), and Sociology (Cohen, 1989). Following are the criteria taken from the work by Sjøberg et al. (2008), where each criterion desginates the degree of support (i.e., low, medium, or high) for the emerging theory from the empirical studies (e.g., [S1], [S2]). The classification of each criterion as low, medium, or high is based on the author's subjective judgment as explained below. Our judgment is derived from the explanation given by Sjøberg et al. (2008) for each criterion. In the following definitions, we only explain high and low classification with the intuition that medium classification would lie in between high and low classification.

1. **Empirical support:** The degree to which a theory is supported by empirical studies that confirm its validity (Sjøberg et al., 2008). 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 if applied in practice.

2. **Utility:** The degree to which a theory supports the relevant areas of the software industry (Sjøberg et al., 2008). 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; whereas, utility of a theory will be low if the theory is not able to reduce the complexity of the empirical world and decision making.

3. **Generality:** The breadth of the scope of a theory and the degree to which the theory is independent of specific settings (Sjøberg et al., 2008). 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.

4. **Testability:** The degree to which a theory can be empirically refuted (Sjøberg et al., 2008). We consider higher testability when propositions of a theory are internally consistent, free from ambiguities, and tested in empirical studies; whereas, 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.

5. **Explanatory Power**: The degree to which a theory accounts for and predicts all known observations within its scope, is simple in that it has few ad hoc assumptions, and relates to that which is already well understood (Sjøberg et al., 2008). We judge that a theory will have high explanatory power if it can be supported by analogies to well known theories and explains all relevant relationships and accounts for all known data in its field; whereas, we consider explanatory power low for a theory if it cannot be associated with well known theories and misses some relationships in its explanation.

6. **Parsimony:** The degree to which a theory is economically constructed with minimum of concepts and propositions. There is a delicate balance between parsiomony and explanatory power. We consider that higher parsimony means removal of unnecessary concepts and propositions (from a theory) that add little additional value to our understanding; whereas, lower parsimony means complex concepts and propositions that are difficult to understand.

Empirical Support

The empirical support of this emerging theory is considered to be medium because the number of programs on which we evaluated F007 (including F007-plus) in the two studies is only 13 small to very large programs, and five of these programs have three to five releases (see Table 4 in Section 2.4.1). If we consider each release as one program then F007 (including F007-plus) was evaluated on approximately 30 programs. This shows that the results were empirically grounded in the results from a sufficient number of programs, and there is certainly room to do more. Thus, we consider that the empirical support is medium.

Utility:

This emerging theory can be used as an aid to maintainers in identifying the origin of faults during corrective maintenance. Maintainers can focus on the traces of 20% of functions/components and can easily use the F007 technique to identify faulty functions in the field failures. We consider utility of this theory to be high. Section 2.6, Section 2.7, and Section 3.8 show the utility of this emerging theory in practical settings.

Generality

Since we have experimented on 13 different programs (see Table 4 in Section 2.4.1) from small to very large sizes, this theory is generalizable to other programs. In 12 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 generalisable 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: (a) F007 analyzed execution traces not the constructs of source code to discover faulty functions; and (b) F007 was evaluated from one to many releases of programs and the results were similar if faulty

traces of functions were present in the training set—i.e., irrespective of the age of a program. Considering all these aspects we judge medium generality for this theory.

Testability

The propositions are defined in a consistent, understandable and non-amibiguous way, atleast from the practitioners and researchers familiar with the topic of this theory. Each of the studies [S1],[S2] can be easily replicated and the stated propositions (P^1_1, P^1_2, P^1_3, P^1_4, and P^1_5) are based on these studies. Each of the propositions has been empirically validated, and they are easily testable. Different study designs (such as identifying faults at system or configuration level) can be used to independently test the propositions. We consider testability high for this emerging theory.

Explanatory Power

The theory presented in this chapter provides an explanation of identification of a faulty function in a failed trace from traces of different faults in that function. This theory also implies that when a small proportion (e.g., 20%) of functions are faulty then the accuracy of identification of faulty functions is high—that is always going to be the case if 20% of the code is responsible for 80% of the faults. However, we think that theory can be made stronger in explanatory power by identifying quantitative characteristics to its attributes; for example: (a) How accurately faulty functions can be identified? (b) What proportion of failed traces can be resolved correctly? (c) Can we generalize this theory quantitiatively like the 80-20 Pareto rule (i.e., using traces of 20% functions can we resolved 80% failed traces!)? Thus, further studies can strengthen the explanatory power of the theory. We consider explanatory power low for this emerging theory.

Parsimony

This emerging theory at both levels of proposition is constructed using few, clear and concise concepts (such as function-calls, traces, faults and faulty functions). An application of these concepts was also shown in the form of F007 (see Section 2.3, Section 2.6, and Section 2.7) and F007-plus (see Section 3.5 and Section 3.8). Thus we think that parsimony is high.

4.5 Implications

There are several implications of this emerging theory on both practice and research:

Practice

- Quality of the software would improve as the maintainers can spend more time on fixing the faults rather than diagnosing the faults.
- Fault locations (e.g., function, component) could also be quickly identified during the "testing" phase of succeeding releases using the failed traces of previous releases.
- Time and effort spent in corrective maintenance would be reduced.
- It could be used in diagnosing faults at a system level or faults in configuration of a system using operating system level call traces.

Research

- 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. 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.
- 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 investigate a new theory using the 80-20 Pareto rule for software code (Boehm and Basili, 2001; Gittens et al., 2005; Ostrand et al., 2005) 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% function?

4.6 Conclusion

In this chapter, we propose an emerging theory based on the empirical findings of the two studies, discussed in Chapter 2 and Chapter 3, of this thesis. This emerging theory identifies that faults in a function can be identified if traces of at least one fault in that function are already known: based on the fact that faults in closely related functions have similar function-call traces.

The emerging theory is stated as: *"A faulty function can be so identified if the traces of at least one fault in that function are already known; and the accuracy of identification increases with the decreasing proportion of faulty functions in the program.*"

The emerging theory was developed in a bottom-up fashion using *hypothetico-inductive* model (Sjoberg et al., 2008) and each of its proposition (see Section 4.2) was empirically grounded in the findings of the two studies (see Section 2.6 and Section 2.7 in Chapter 2, and Section 3.8 in Chapter 3). Subsequently, we also evaluated this theory (see Section 4.4) on the basis of "theory goodness" criteria proposed by Sjoberg et al. (2008) and similar criteria of Boehm and Jain (2005). This theory is still in initial stages as it was done on only thirteen programs of small to very large sizes. Overall, the emerging theory satisfies the goodness criteria of utility, generality, parsimony, testability, empirical support and explanatory power (see Section 4.4).

Clearly, more empirical studies are needed to test in detail the specific aspects of the theory (such as what are those functions that form a group and have same function-call traces). Thus, more efforts are needed from the maintenance community to conduct studies in various contexts and from various perspectives to strengthen this emerging theory.

4.7 References

Bacharach, S.B. "Organizational theories: some criteria for evaluation." *The Academy of Management Review*, Vol. 14, No. 4, Oct., 1989, pp. 496–515.

Boehm, B. & Basili V., R. "Software Defect Reduction Top 10 List", *Computer,* Vol. 34, No. 1, IEEE CS Press, Jan. 2001, pp. 135-137.

Boehm, B.; and Jain., A."An initial theory of value-based software engineering," in *Value-Based Software Engineering*, 1st Edition. Biffl, S.; Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher G., Eds., Springer, LNCS, Germany, 2005, pp. 15-33.

Carroll, J. and Swatman, P.A."Structured-case: a methodological framework for building theory in information systems research." *European Journal of Information Systems*, Vol. 9, No. 4, Dec., 2000, pp. 235–242.

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

Do, H., Elbaum, S. G., and Rothermel, G."Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." *Empirical Softw. Eng.*, Vol. 10, Springer, Oct. 2005, pp. 405-435.

Ferrari, R. "An emerging theory on the interaction between requirements engineering and systems architecting based on a suite of exploratory empirical studies." Ph.D. thesis, University of Western Ontario, Sep., 2010.

Gittens M.; Kim Y.; and Godwin D. "The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software." Proc. 29th *Int'l Conf. Computer Softw. and Appl.*, Edinburgh, Scotland, July 2005, pp. 179-185.

Haig, B.D. "An abductive theory of scientific method." *Psychological Methods*, Vol.10, No. 4, 2005, pp. 371–388.

Hutchins, M.; Foster, H.; Goradia, T.; Ostrand, T., "Experiments on The Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria". *Proc. 16th Int'l Conf. on Softw. Eng.*, IEEE, Sorrento, Italy, May,1994 ,pp.191-200

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

Offutt, A., J. and Untch, R., H., "Mutation 2000: uniting the orthogonal," in *Mutation Testing for the New Century*, Wong W,E., Ed., USA: Kluwer Academic Publishers, 2001, pp. 34-44.

Ostrand T. J.; Weyuker E. and Bell R. M. "Predicting the Location and Number of Faults in Large Software Systems*." IEEE Trans. on Softw. Eng*., Vol. 31, No. 4, 2005, pp. 340-355.

Sjøberg, D.; Dyba, D.; Anda, B. C.; and Hannay. J. "Building theories in software engineering," in *Guide to Advanced Empirical Software Engineering*. Shull,F., Singer,J., and Sjøberg,D.I.K, Eds., London: Springer, 2008, pp. 312–336.

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

Chapter 5

5 Conclusions and Future Work

In this section, we conclude this thesis in Section 5.1 by reflecting on the two studies of this thesis and the emerging theory derived from those two studies. Finally, in Section 5.2 we present the future work.

5.1 Conclusions

Corrective software maintenance (which deals with the correction of faults) can soak up to approximately 30-60% time of software maintenance activities (Schach et al., 2003; Lee and Jefferson, 2005). One of the time consuming and difficult activities of corrective maintenance is identification of the origin of fault that can consume approximately 30-40% time of corrective maintenance (Proprietary Workshop, 2008). To aid in reducing the time and effort spent in corrective maintenance this thesis addresses the problem of automatically finding the finer-grained fault locations (faulty function) in the execution traces of field failures.

The solution to this problem is proposed in the form of the technique F007 that automatically identifies faulty functions from the function-call level execution traces of field failures. F007 predicts a ranked list of faulty functions for a failed trace by training the decision trees on the historical collection of failed traces. This thesis incorporates two studies: (a) first study proposes F007 and shows that F007 can be used to identify rediscovered field faults; and (b) second study proposes F007-plus that improves F007 by showing that F007 can be used to discover both new and rediscovered faults. Each study is documented in its own chapter with its own introduction, related work, methodology, evaluation and conclusions.

In the first study, F007 identifies faulty functions in a field trace of the latest release of a program by training the decision tree on: (a) the historical collection of failed traces of the same release of a program; and (b) the failed traces of preceding releases and the latest release of a program (see Chapter 2). Though F007 is useful--especially when 50-

90% failures are rediscoveries of the same fault (Brodie et al., 2005; Lee and Iyer, 2000; Wood, 2003) and 20% of the code is responsible for 80% of the faults (Gittens et al., 2005; Ostrand et al., 2005)--a limitation of F007 in first study is that it can identify those faulty functions that are observed in previous traces of actual faults; new faulty functions can not be identified.

The second study (see Chapter 3) overcomes this limitation by proposing a new strategy F007-plus. F007-plus trains F007 on the failed traces of mutants (artificial faults) to identify faulty functions in the actual field traces. F007-plus also uses the failed traces of previous and current releases, if any, in training F007. The use of mutants facilitates F007 to identify new faulty functions.

We evaluated F007 and F007-plus on thirteen subject programs with several releases; that is: (i) seven programs of the Siemens suite with one release (Do et al., 2005), (ii) the Space program with one release (Do et al., 2005), (iii) the Flex program with five releases (Do et al., 2005), (iv) the Grep program with four releases (Do et al., 2005), (v) the Gzip program with four releases (Do et al., 2005), (vi) the Sed program with five releases (Do et al., 2005), and (vii) a very large commercial software application (of 20 million LOC) with three releases. Our findings from the two studies are:

(a) Patterns of function-calls do not yield better fault identification accuracy than the single function-calls when used with the classification algorithm such as the decision tree (see Chapter 2, Section 2.6.1).

(b) The size of a function-call level (see Figure 1, chapter 1) trace can be reduced to half because only “function entry” or ‘function exit” events are enough to identify fault locations in a failed trace. Using both of them (“function entry” and “function exits”) together has no effect on the accuracy of identification of the fault (see Chapter 2, Section 2.6.4). This discovery could also decrease the runtime overhead of a function-call trace collection to approximately half.

(c) A group ‘Mi’ of related functions have similar function-call traces when a fault occurs in the functions of that group ‘Mi’; but the 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 $i \neq k$ and $Mi \subset N$ and $Mk \subset N$ and N={functions | functions $\in$ program (see Chapter 3, Section 3.7).

(d) In general, F007 (including F007-plus) can identify faulty functions in approximately a maximum of 90% failed traces of the subject programs on reviewing 10% or less of the source program (different settings result in different results; see, for example: Section 2.6.2 and Section 2.7; and Section 3.8).

We also observed that the above findings of this thesis lead to the foundation of an emerging theory. We, therefore, proposed an emerging descriptive theory that is empirically grounded in the findings of two studies shown of this thesis (see Chapter 4), and is stated as:

"A faulty function can be so identified if the traces of at least one fault in that function are already known; and the accuracy of identification increases with the decreasing proportion of faulty functions in the program."

Although, the above results are encouraging, but except for only one very large industrial program in the first study (see Chapter 2, Section 2.7), all the experiments were conducted in the lab settings. Therefore, more experiments on the programs with actual field settings are required to be done.

5.2 Future Work

In Chapter 3, we identified expected faulty functions of a current release using the code metrics of previous releases. The prediction of expected faulty functions, which was done using cost sensitive learning, included many false positives (see Section 3.5). Another future research issue is to explore this further to identify exactly right set of expected faulty functions and their mutants for training. This will further improve the accuracy of identification of faulty functions in the actual traces, and will result in the reduced effort of mutants and failed traces generation.

Another possible research area is the application of cost sensitive learning in the identification of faulty functions in the failed traces. If cost sensitive learning strategy is used in training the decision tree in F007 then the important research issue will be to develop the criteria to assign costs to faulty functions. For example, functions or components with highly critical functionality can be assigned higher cost. This would result in high accuracy of identification if fault occurs in those highly critical functions, but would result in low accuracy on the non-critical functions.

Due to widespread use of mobile applications, the application of F007 on mobile systems is a potential area of research. An important research issue is to use execution traces containing limited information due to small processing power of mobile systems and accurately identify a fault's origin.

In distributed systems or multiple software systems that run together (e.g., web server running with database management system), traces contain sequence of events instead of sequence of function-calls. An important research issue is to determine whether the traces of events follow similar patterns as function-call traces as identified by F007 (i.e., different faults in a function have similar traces). This will help in quickly identifying a fault's origin at the configuration level of a system.

Finally, there are certain issues in commercialization of the F007 technique and its F007-plus strategy. For example, a framework or integrated development environment is needed that can allow practitioners to automatically identify faulty functions in failed traces, automatically identify expected faulty functions in a current release from the data of past releases, generate mutants automatically for the expected faulty functions and collect mutant traces for the program. If these steps can be automated or semi-automated, only then the chances are high for the impact of F007 technique on practice.

5.3 References

Brodie, M.; Sheng Ma; Lohman, G.; Mignet, L.; Modani, N.; Wilding, M.; Champlin, J.; Sohn, P., "Quickly Finding Known Software Problems via Automated Symptom Matching", *Proc. of Second Int'l Conf. on Autonomic Computing, ICAC 2005*, IEEE CS, June 2005, pp. 101-110.

Do, H., Elbaum, S. G., and Rothermel, G. “Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.".*Empirical Softw. Eng*., Vol. 10, Springer, Oct. 2005, pp. 405-435.

Gittens M., Kim Y., and Godwin D., “The vital few versus the trivial many: Examining the pareto principle for software”, *Proc. of 29th Int'l Computer Software and Applications Conf. (COMPSAC'05)*, IEEE CS, Edinburgh, Scotland, July 2005, pp. 179-185.

Lee M. G. and Jefferson T. L. “An Empirical Study of Software Maintenance of a Web-based Java Application.” *Proc. Int’l Conf. on Softw. Maintenance*, IEEE, Budapest, Hungary, Sep., 2005, pp. 571-576.

Lee, I.; Iyer, R., “Diagnosing Rediscovered Problems Using Symptoms”, *IEEE Transactions on Software Engineering*, vol. 26, no. 2,IEEE USA, Feb. 2000, pp. 113-127.

Ostrand T. J., Weyuker E., and Bell R. M., “Predicting the Location and Number of Faults in Large Software Systems.” *IEEE Trans. on Softw. Eng*., Vol. 31, No. 4, 2005, pp. 340-355.

Proprietary workshop on large commercial software, Sep., 2008.

Schach S. R.; Jin B.; Yu L.; Heller G. Z.; and Offutt J. “Determining the Distribution of Maintenance Categories: Survey versus Measurement.” *Empirical Softw. Eng*., Vol. 8, No. 4, Springer, Dec., 2003, pp. 351-365.

Wood A., “Software reliability from the customer view”, *Computer* , vol. 36, no. 8, IEEE CS, USA, August 2003, pp. 37-42

Appendix

Detailed Results of F007 using the MINEPI Algorithm and Function Entry and Function Exit Calls

In this section, we show the results of F007 using patterns of function-calls (i.e., the MINEPI algorithm (Mannila et al., 1997)) and single function-calls on all the subject programs. The Shapiro-wilk test and the Wilcoxon signed rank tests (Marques de Sá, 2003) in (Chapter 2) Section 2.6.1.1 were conducted on this dataset.

Table 17: Accuracy of F007 using the patterns of function-calls (the MINEPI algorithm) and using only single function-calls.

Program	Episode Rule Type	Length	Win (w) = 3	Win(w) = 5	Win (w) = 7
Print_tokens	NA	1	74.380	74.380	74.380
	Serial/Parallel	2	73.347	68.801	70.041
	Serial	3	73.553	72.933	70.041
	Parallel	3	73.347	71.900	68.595
Print_tokens2	NA	1	61.773	61.773	61.773
	Serial/Parallel	2	56.298	57.364	55.523
	Serial	3	56.346	57.655	58.624
	Parallel	3	57.9	57.509	57.46
Replace	NA	1	65.447	65.447	65.447
	Serial/Parallel	2	65.963	65.611	64.932
	Serial	3	64.861	65.096	65.518
	Parallel	3	63.97	63.806	64.838
Schedule	NA	1	73.248	73.248	73.248
	Serial/Parallel	2	65.478	67.643	67.643
	Serial	3	66.930	64.698	64.698
	Parallel	3	66.667	64.041	64.042
Schedule2	NA	1	60.363	60.363	60.363
	Serial/Parallel	2	65.0909	61.454	62.909
	Serial	3	60.0	62.181	60.727
	Parallel	3	62.909	62.181	59.636
Tcas	NA	1	73.481	73.481	73.481
	Serial/Parallel	2	73.546	73.546	73.416
	Serial	3	73.807	73.807	73.416
	Parallel	3	72.045	72.045	73.807
Tot_info	NA	1	68.842	68.842	68.842
	Serial/Parallel	1,2	66.831	66.666	67.325
	Serial	1,3	65.831	66.602	65.960
	Parallel	1,3	60.244	63.326	65.575
Space	NA	1	80.425	80.425	80.425
	Serial/Parallel	2	74.333	74.410	75.324

	Serial	3	75.124	75.124	74.215
	Parallel	3	75.721	75.321	76.100
Grep (R3)	NA	1	95.546	95.546	95.546
	Serial/Parallel	2	99.595	98.380	98.380
	Serial	3	99.190	97.571	97.976
	Parallel	3	98.785	97.976	98.380
Sed (R3)	NA	1	89.361	89.361	89.361
	Serial/Parallel	2	95.745	93.617	93.617
	Serial	3	95.035	94.326	94.326
	Parallel	3	94.326	95.035	92.198
Gzip (R1)	NA	1	90.0	90.0	90.0
	Serial/Parallel	2	90.0	90.0	90.0
	Serial	3	92.0	92.0	90.0
	Parallel	3	92.0	90.0	90.0
Flex (R3)	NA	1	58.563	58.563	58.563
	Serial/Parallel	2	61.050	61.050	61.602
	Serial	3	61.326	60.773	61.602
	Parallel	3	61.602	61.050	60.773

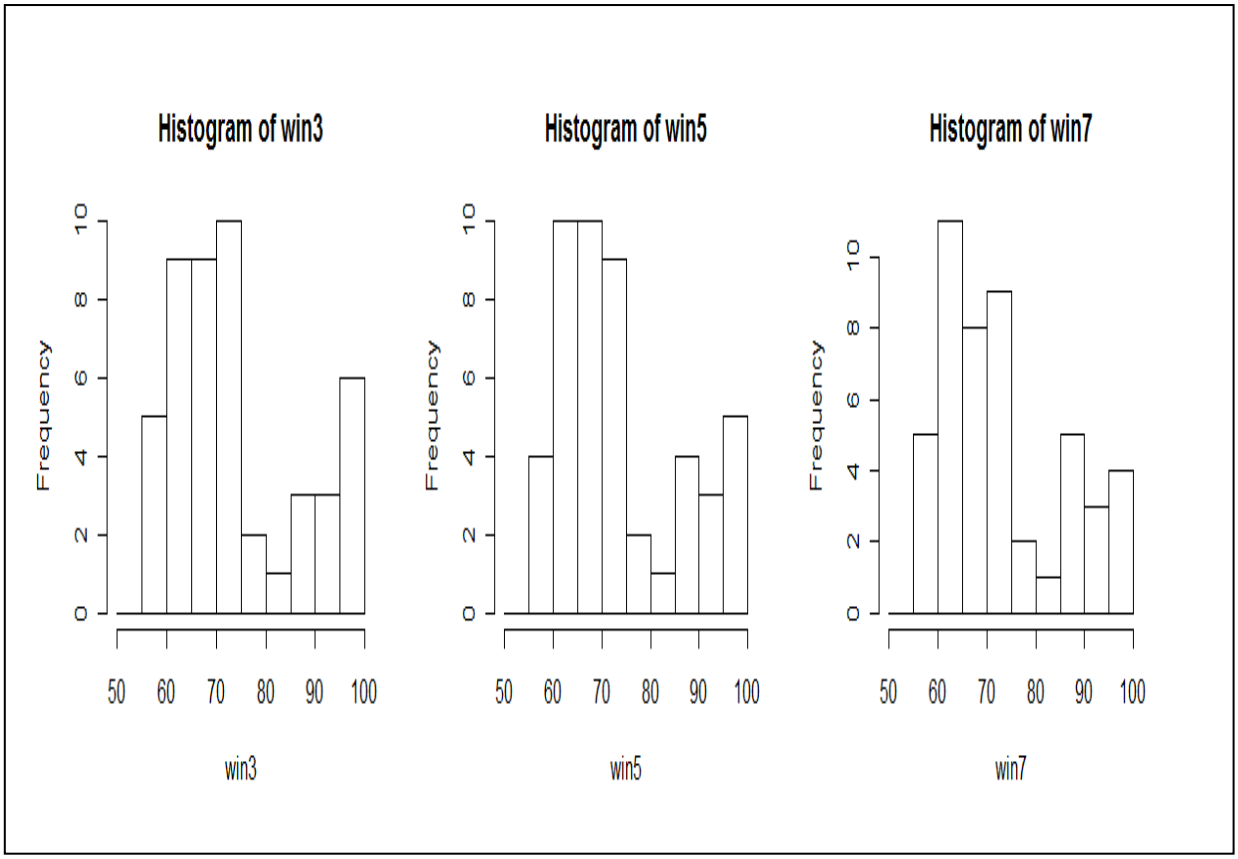


Figure 42: Histograms of different accuracies at win(w)=3, win(w)=5 and win(w)=7 using bin of 5 (percentage) units.

In order to determine whether the dataset is normal, we first conducted the Shapiro-wilk test on the accuracy values of the twelve subject programs corresponding to each of the window widths in Table 17. The Shapiro-wilk test on 48 data points at: (a) win(w)=3 resulted into W=0.887 and p=0.00024; (b) win(w)=5 resulted into W=0.876 and p=0.00011; and (c) win(w)=7 resulted into W=0.8843 and p=0.0002. This means that data distribution is not normal for each of the window widths because p value is less than alpha value of 0.05 ($p < 0.05$) for each of the window widths. To further investigate about the distribution of data, we constructed the histogram of data points for each of the window widths, which are shown in Figure 42. These histograms show positively skewed data distributions, which again confirm the non-normal distribution of the data points.

Thus, we conducted the Wilcoxon signed rank test as the data distribution is not normal. We first conducted the Wilcoxon signed-rank test between the window width 5 and the window width 7 with the null hypothesis: "*there was no significant difference between classification accuracy of* win(w)=*5 and* win(w)=*7*". We again set the alpha level to 0.05 as at this level we can reduce the risk of type 1 error (false positive).The Wilcoxon signed rank test for 48 (i.e., for 12 programs) matched observations did not result in significant difference between "win(w)=5" (M=73.50, SD=12.998) and "win(w)=7" (M=73.27, SD=12.983) with z=0.803 and (two-tailed) p=0.422. This provided the evidence ($p > 0.05$) that the null hypothesis could not be rejected and the fault prediction accuracies of "win(w)=5" and "win(w)=7" were identical.

Similarly, the Wilcoxon signed-rank test between: (a) "win(w)=3" (M=73.80, SD=13.299) and "win(w)=5" (M=73.50, SD=12.998) produced z = 1.860 (two tailed) p=0.63; and (b) "win(w)=3" (M=73.80, SD=13.299) and "win(w)=7" (M=73.27, SD=12.983) generated z=2.326 (two-tailed) p=0.020. In the case of win(w)=3 and win(w)= 5 (case 'a'), the accuracies with different window widths were identical (i.e., $p > 0.05$). In the case of win(w)=3 and win(w)=7, the accuracies were not identical (i.e., $p < 0.05$); the accuracy to discover the faulty functions at win(w)=7 had actually started decreasing.

We can also observe from Table 17 at win(w)=3 that accuracy also varies among the episode lengths. To determine if there is any improvement in the accuracy between different episodes rules, we conducted the Wilcoxon signed rank test between episodes of length 1 and the higher episode rules within win(w)=3. The Wilcoxon signed rank test with 12 observations between: (a) the episodes of length 1 (M=74.285, SD=12.325) and the serial/parallel episode rules (M=73.940, SD=13.927) of length 2 resulted in z=0.178 and (two tailed) p=0.859; (b) the episodes of length 1(M=74.285, SD=12.325) and the serial episode rules of length 3 (M=73.667, SD=14.364) resulted in z=0.628 and (two tailed) p=0.530; and (c) the episodes of length 1 (M=74.285, SD=12.325) and the parallel episode rules of length 3 (M=73.293, SD=14.245) yielded z=0.549, (two tailed) p=0.583.

In all these cases the value of p is significantly higher than significance level of 0.05, substantiating that there is no significant difference between the accuracy of the episodes of length 1 and the higher length (serial or parallel) episode rules. This implies that the episodes of length 1, which are just single function-calls, are not only cost-effective to generate, but also yield equivalent (or better) fault prediction accuracy than the higher length episode rules.

Table 18: Accuracy of identifying faulty functions using the episodes of length 1 with frequency and confidence.

Program	Frequency	Confidence
Print_tokens	73.967	74.380
Print_tokens2	55.475	61.773
Replace	66.083	65.447
Schedule	71.337	73.248
Schedule2	61	60.363
Tcas	73.481	73.481
Tot_info	68.053	68.842
Space	80.529	80.425
Sed	93.617	89.361
Gzip	90.0	90.0
Grep	98.785	95.546
Flex	58.287	58.563

In Table 17, we used the confidence values with the episodes of different lengths to identify the faulty functions using the decision trees. Table 18 shows the accuracy of the identification of the faulty functions using the frequency and the confidence values with

the episodes of length 1. We gain conducted the Wilcoxon singed-rank test to determine if there was any difference between the accuracies of the identification of the faulty functions using the frequency values or the confidence values with the episodes of length 1. The Wilcoxon signed-rank test with 12 observations resulted into z=0.178 and (two-tailed) p=0.859. This shows that when decicion tree is used there is no significant difference between the frequency and confidence values of the episodes of length 1 (or simply function-calls). Thus, either frequency or confidence can be used with the function-calls to identify the faulty functions in the failed traces using the decision trees.

Table 19: Accuracy of F007 using only function "entry or exit" or both function "entry and exits".

Program	Episode Rule	Entry and Exits	Entry or Exits
Print_tokens	1	75.206	74.3801
	(S/P) 2	70.867	73.553
	(S) 3	73.14	73.55
	(P) 3	73.347	73.347
Print_tokens2	1	60.804	61.77
	(S/P) 2	58.527	56.298
	(S) 3	59.011	56.346
	(P) 3	60.222	57.9
Replace	1	64.791	65.447
	(S/P) 2	66.198	65.963
	(S) 3	66.268	64.861
	(P) 3	66.057	63.97
Schedule	1	73.248	74.267
	(S/P) 2	65.061	65.350
	(S) 3	64.667	65.605
	(P) 3	66.388	66.369
Space	1	80.425	80.425
	(S/P) 2	76.577	74.333
	(S) 3	75.124	75.124
	(P) 3	75.721	75.721
Schedule2	1	60.363	60.363
	(S/P) 2	68.0	65.0909
	(S) 3	63.272	60
	(P) 3	59.272	62.909
Tcas	1	73.481	73.481
	(S/P) 2	73.873	73.546
	(S) 3	73.481	73.807
	(P) 3	73.481	72.045
Tot_info	1	68.842	68.842
	(S/P) 2	68.947	66.831

	(S) 3	67.819	65.831
	(P) 3	63.207	60.244
Flex (R3)	1	58.840	58.563
	(S/P) 2	63.260	58.563
	(S) 3	60.773	61.050
	(P) 3	60.773	61.326
Grep (R3)	1	100.0	95.546
	(S/P) 2	99.190	99.595
	(S) 3	99.595	99.190
	(P) 3	99.190	98.785
Sed (R3)	1	89.361	89.361
	(S/P) 2	92.908	95.745
	(S) 3	92.908	95.035
	(P) 3	92.908	94.326
Gzip (R1)	1	90.0	90.0
	(S/P) 2	90.0	90.0
	(S) 3	86.0	92.0
	(P) 3	90.0	92.0

In all of the above results, we have either used only function “entry” OR function “exit” symbols. In the end, in this appendix we also show the accuracy of the identification of the faulty functions using both the function “entry and exits” and only the function “entry or exits” on all the programs (see Section 2.6.4). This is shown in Table 19, where first column shows the name of the program, second column shows the accuracy obtained using both the function “entry and exits” and the last column shows the accuracy using only the function “entry or exits”. In Table 19, for function “entry or exits”, we have randomly selected for some programs only function “entry” symbol and for some programs only function “exit” symbols. Furthermore, the results in Table 19 were obtained using the confidence values for the function “entry and exit” and “entry” or “exits”, and using win(w)=3.

The Wilcxon signed-rank test between the function “entry and exits” (M=73.98, SD=12.770) and only the function “entry or exits” (M=73.721, SD=13.656) resulted into z=0.837 and p=0.402; substantiating that there is no significant difference between the accuracies of the identification of the faulty functions using function “entry and exits” and only function “entry or exits”. Thus function “entry or exits” are sufficient to discover the faulty functions because they result in approximately 50% lesser trace size, reduced overhead in the trace collection and storage of the traces.

References

Mannila, H.; Toivonen, H; Inkeri,V.; "Discovery of Frequent Episodes in Event Sequences". *Data Mining and Knowledge Discovery*, Vol. 1, No. 3, Springer, Jan 1997, pp. 259-289.

Marques de Sá, J., P.; *Applied Statistics Using SPSS, STATISTICA, MATLAB and R*, 1st ed., Springer, Aug., 2003.

Glossary of Terms

In this section, we present a glossary of important terms used in this dissertation:

Adaptive software maintenance: Adaptive software maintenance is defined as the type of software maintenance which allows programs to work in a changed environment (IEEE Std. 610.12, 1990; Swanson, 1976); for example, adapting to a new operating system.

Corrective maintenance: Corrective maintenance is the software maintenance which corrects faults of a system (IEEE Std. 610.12, 1990; Swanson, 1976); for example, the corrective maintenance activity corrects functional or processing faults reported by users.

Cost sensitive learning: In cost sensitive learning a classifier (e.g., decision tree) is forced to make lesser error on one type of category (e.g., faulty functions) of dependent variable and more errors on other type of category (e.g., not-faulty functions) of dependent variable (Witten and Frank, 2005).

Decision tree: The decision tree is a machine learning classification algorithm, which specify sequences of decisions (from the independent attributes of a dataset) that need to be made for a particular outcome (dependent attribute of the dataset). Formally, it is a method of approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions (Mitchell, 1997).

Execution trace: An execution trace is a record of the sequence of code labels (e.g., statements, functions) executed during a particular run of a program (IEEE Std. 610.12, 1990).

Failure: The inability of a system or component to perform its required functions within specified performance requirement (IEEE Std. 610.12, 1990).

Fault: An incorrect step, process or data definition in a computer program (IEEE Std. 610.12, 1990).

Mutants: Mutants are automatically seeded faults generated by changing statements of a program (Offutt et al., 2001).

One-against-all approach: In this approach, a dataset with M categories of dependent variable is decomposed into M new datasets with binary categories. Each new binary dataset 'Di' has category 'Ci' (where i = 1 to M) labeled as positive and all other categories are labeled as negative. On each new datasets 'Di' the decision tree algorithm is trained; resulting in 'M' trees in total. Each decision tree predicts its category 'Ci' of the dependent variable along with a probability of being faulty. The category 'Ci' with the highest probability is considered faulty (Witten and Frank, 2005). Another way of finding the faulty category 'Ci' is that the category 'Ci' predicted by most decision trees is considered faulty.

Perfective maintenance: Perfective maintenance is the type of software maintenance which is done to improve the maintainability, performance, or other quality attributes of a software system (IEEE Std. 610.12, 1990; Swanson, 1976); for example, adding new features.

Software maintenance: Software maintenance is defined as the process of modifying a software system or a component after its delivery to customers (IEEE Std. 610.12, 1990).

Types of software maintenance: Software maintenance was divided into three different categories by Swanson (1976): adaptive, perfective and corrective. Each of them is explained separately in this glossary.

References

IEEE Std. 610.12, *Standard Glossary of Software Engineering Terminology*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

Mitchell T. M., *Machine Learning*, McGraw Hill, 1997.

Offutt, A., J.; and Untch, R., H. "Mutation 2000: Uniting the Orthogonal," in *Mutation Testing for the New Century*, Wong W,E., Ed., USA: Kluwer Academic Publishers, 2001, pp. 34-44.

Swanson E. B., "The Dimensions of Maintenance", *Proc. of 2nd Int'l Conf. on Softw. Eng., IEEE CS*, San Francisco, USA, Oct., 1976, pp. 492-497.

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

Curriculum Vitae

Name:	**Syed Shariyar Murtaza**
Preferred Name:	Shariyar
Post-secondary Education and Degrees:	The University of Karachi Karachi, Sind, Pakistan B.S. (Computer Science)--2004 Kyung Hee, University Suwon, Gyeonggi-do, South Korea M.S. (Computer Science)--2006 The University of Western Ontario London, Ontario, Canada Ph.D. (Computer Science)--2011
Honours and Awards:	Province of Ontario Graduate Scholarship 2009-2010 Korean govt. (IITA) MS Scholarship 2004-2006 University of Western Ontario Graduate Teaching and Research Assistance Award 2006-2010 Best presenter award at several venues.
Related Work Experience	Teaching and Research Assistant The University of Western Ontario, 2006-2008 Research Assistant Kyung Hee University, South Korea, 2004-2006 Software Developer Perheal Pvt. Ltd., Hodgins Media Ventures Inc. 2003-2004, 2007-2008

Publications:

[1] **Murtaza, S.S**.; Madhavji, N.H.; Gittens, M.; Li, Z.; "Diagnosing New Faults Using Mutants and Faults of Prior Releases (NIER Track)", Proc. of 33rd *International Conference on Software Engineering*, ACM, Honolulu, Hawaii, USA, May, 2011-- *accepted*.

[2] **Murtaza, S.S.**; Gittens, M.; Li, Z; Madhavji, N.H.: "Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field", Proc. 2010 *conference of the Centre for Advanced Studies on Collaborative Research*, ACM, Nov. 2010, Canada.

[3] Li, Z.; Gittens, M.; **Murtaza, S.S.**; Madhavji, N.H.; Miranskyy, A.V.; Godwin, D.; and Cialini, E.: "Analysis of Pervasive Multiple-Component Defects in a Large Software System", proc. of 25th IEEE *International Conference on Software Maintenance (ICSM)*, IEEE, Edmonton, Alberta, Canada, Sep. 2009, pp. 265-273.

[4] **Murtaza, S.S.**; Gittens, M.; Madhavji, N.H.: "Discovering the Fault Origin from Field Traces", Proc. of 19th *International Symposium on Software Reliability Engineering*, IEEE CS, Seattle, USA, Nov. 2008, pp. 295-296.

[5] **Murtaza, S.S.**; Ahmed, B.; Hong, C.S.:" **On the Dynamic Management of Information in Ubiquitous Systems using Evolvable Software Components**", Proc. of 9th Asia *Pacific Network Operation and Management Symposium (APNOMS)*, Springer Verlag, LNCS, Okinawa, Japan, Sep. 2006, pp. 513-516.

[6] **Murtaza, S.S.**: **MS Thesis**: "**A Dynamic Ontology Based Ubiquitous System using Evolvable Software Components", June 2006**. It was an autonomic and ubiquitous system, which learnt the user schedule and preferences when it interacted with the UPnP based devices. The system then automated the environment based on its learnt knowledge about the user. All the devices (TV, Fridge, Light Bulb, etc.) along with the surveillance and controlling applications were developed using C# and Intel UPnP SDK.

[7] **Murtaza, S.S.**; Hong, C.S.:" **A Conceptual Architecture for Uniform Identification of Objects**", Proc. of the *Fourth Annual ACIS International Conference on Computer and Information Science (ICIS' 05),* IEEE CS, July. 2005, pp. 670-675.

[8] **Murtaza, S.S.**; Hong, C.S.:" **An Evolvable Software Architecture for Managing Ubiquitous Systems**", Proc. of the 8th *Asia Pacific Network Operation and Management Symposium (APNOMS*), Sep. 2005, pp. 400-409.

[9] **Murtaza, S.S.**; Amin S.O.; Hong C.S.: "**Applications of SNMP in Ubiquitous Environment**", KNOM Review, Journal, Vol. 8, No. 2, Feb. 2006, pp. 14-19.

[10] **Murtaza, S.S**.: **Internet draft** on IETF, "**A Web of Physical Objects' Uniform Resource Identifier**", March 2, 2004, http://ietfreport.isoc.org/all-ids/draft-shariyar-wop-uri-01.txt.

[11] **Murtaza, S.S.**: **BS Thesis: "A Web of Physical Objects", Dec 2003.** It analyzed the trends and practices of semantic web and ubiquitous computing. It also proposed a URI scheme for physical objects and was published as an internet draft on IETF.