2010 IEEE 21st International Symposium on Software Reliability Engineering

Empirical Evaluation of Factors Affecting Distinction Between Failing and Passing Executions

Arin Zahalka, Katerina Goševa-Popstojanova, Jeffrey Zemerick
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506, USA
{azahalka, jzemeric}@mix.wvu.edu, katerina.goseva@mail.wvu.edu

***Abstract*—Information captured in software execution profiles can benefit verification activities by supporting more cost-effective fault localization and execution classification. This paper proposes an experimental design which utilizes execution information to quantify the effect of factors such as different programs and fault inclusions on the distinction between passed and failed execution profiles. For this controlled experiment we use well-known, benchmark-like programs. In addition to experimentation, our empirical evaluation includes case studies of open source programs having more complex fault models. The results show that metrics reflecting distinction between failing and passing executions are affected more by program than by faults included.**

I. Introduction

In-house pre-release testing often involves running a large collection of tests as part of the process of uncovering faults. In the early stages when software still contains a multitude of faults, a large number of tests may fail of which many can be non-crashing, and therefore not obvious failures. In the absence of an oracle, whether a test fails or passes may require manual inspection, which for very large test suites can become very cumbersome and time-consuming. Regression testing may also share some of the same difficulties: regression test suites often grow very large over many releases. To further exacerbate the process, debugging the failures observed is also time-consuming and difficult. Thus, automating any part of the testing and bug-fixing process can save a great deal of time and effort.

Some of the work conducted in automating the testing process involves leveraging dynamic control flow information. Different types of control flow have been utilized to prioritize and minimize test suites [8], [11], classify executions as passed or failed [2], [10], [7], classify failed executions by fault [18], [15], and identify possible fault locations in source code [13], [12], [19], [23], [3]. The methods and approaches proposed in these studies, either explicitly or implicitly, rely, at least in part, on the assumption that the distinction between a failed and a passed execution is reflected in the control flow. The empirical results of some studies indicate in a qualitative way that "failures often have unusual profiles that are revealed by cluster analysis" [5], [6], while other studies present accurate pass/fail classification of executions using control flow profiles [2], [10].

Motivated by the assumption that control flow profiles reflect some degree of distinction between failed and passed executions, this study explores the *behavior* of this distinction under specific factors. In particular, we employ a two-factorial design where we treat programs and faults as the two factors (independent variables) that we manipulate. This allows us to study the effect of including a various number of faults. For each program under consideration we control the number of faults injected into the code and call this factor the *fault-inclusion* factor. Note that we do not control the inherent type or location of faults. Rather, in our experiments faults are randomly selected from the existing faults available for each particular program.

Focusing on control flow execution profiles at function level (as this level of information capture has been shown to be effective [10], [14], [16]), we use clustering as a natural way to group similar executions together. We propose a set of distinction metrics which measure different aspects of the results of the groupings (i.e., clusters) and use a combination of controlled experimentation and case studies to explore how these clustering distinction metrics are affected by varying fault-inclusion levels and/or programs. In this context, we further refine our main goal with the following research questions:

- RQ1 *Do varying programs have varying distinction metrics? What is the magnitude of the effect programs have on distinction metrics and is it statistically significant?*
- RQ2 *Do increasing fault-inclusion levels per program affect the distinction metrics? What is the magnitude of this effect and is it statistically significant?*
- RQ3 *Do distinction metrics display the same (increasing or decreasing) trend across different programs as the fault-inclusion levels per program increase?*

The main contributions of this paper include:

- We propose a set of metrics which provide different views on the clusters of program execution profiles and collectively allow us to *quantify* and *compare* the extent to which execution profiles of failed test cases are distinct from the execution profiles of passed test cases. It should be noted that these metrics are general, as they do not depend on the type or granularity of the

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.44

profile, nor do they depend on the clustering method.

- We develop an experimental framework based on a well-defined *experimental design* that can be used with different execution profiles, clustering techniques, and response variables. Exploring our research questions by means of formal experimental design allows us (1) to control the factors with an explicit goal to distinguish between the influence of different programs and the influence of their corresponding fault-inclusion levels on pass/fail clustering distinction; (2) to assess quantitatively the *contribution* of each factor on pass/fail distinction; and (3) to conduct the analysis and evaluation in a *statistically* sound context, including testing the statistical significance of the results.
- To the best of our knowledge, this is the first study to *systematically* explore and quantify the pass/fail distinction of clustering function level control flow profiles under varying fault-inclusion levels. This aspect of our work, at least partially, was motivated by several earlier studies (e.g., [10], [13]) indicating multi-fault versions may degrade the results of proposed methods compared to single-fault versions of the same program.
- Our empirical investigation combines a controlled experiment with *case studies* based on two open source programs. Both case studies have more complex fault models and one of them is significantly larger than the programs used as experimental objects. This allows us to gauge the distinction between failing and passing executions in more complex settings and on a larger scale, which typically is not feasible in controlled experiments either because it is associated with an excessive cost or because there is a lack of information to allow the level of control required by experimentation.

The rest of this paper is organized as follows. Related work is presented next. In section III we introduce the notation and proposed distinction metrics. Details of the experimental design are presented in section IV, followed by a description of the empirical objects in our study. Section VI presents the analysis of our results, and section VII presents the threats to validity. Section VIII concludes the paper.

II. Related Work

There has been much work capitalizing on control flow execution events. One area is fault localization. The Tarantula fault-debugging visualization system [13] utilizes the differences between the control flows of failed and passed executions to rank suspicious parts of a program. Tarantula was tested on single-fault versions and multiple-fault versions of the Space program (∼6K LOC, with injected, real faults). Results with multiple faults (up to five) showed decreased effectiveness. The study in [24] also experimented with multi-fault space versions to determine the effect of test-suite reduction on fault localization; though, faults were not the focus. Several other fault localization studies ([19], [12], [23], among others) have also utilized control flow and have been compared to Tarantula using the Siemens software which consists of multiple single-fault versions of seven programs, each less than 1K LOC.

In [18], control flow at the function level was used to group together executions failing due to the same fault. Another study [15] proposed a failure indexing model to encompass such techniques that find which failures are due to the same fault. Numerous other studies used control flow of test executions to help prioritize test cases, usually for regression testing, so that faults are revealed earlier in the testing process, including [8], [1], [11].

Another group of studies utilized control flow for execution pass/fail classification. In [2], control flow information was used to build Markov models of executions to train a classifier on single-fault versions of the Space program. In [10], they found that control flow captured at the coarse granularity of function visit counts made a 'near-perfect' predictor for pass/fail classification. Execution outcomes of single-fault versions of a medium size program (60K LOC) were classified with nearly 100% accuracy. However, multiple-fault versions (two to six faults) had up to 18% reduced accuracy.

In another group of studies [5], [6], the execution profiles at various granularities (including function level) and of various software products (ranging from 3K LOC to 300K LOC) were clustered. The goal was to leverage the groupings so as to find failed executions while reducing the number of total executions examined in the process. It was found that failures were often located in a different area of the execution space than passed executions and that failures were often each assigned to their own cluster. Based upon these observations, several test selection methods were proposed and evaluated. Function-level information was found to be effective in these studies as well as in subsequent studies [16]. Our study is close to these studies in that we also cluster execution profiles with a focus on the way failed executions cluster. However, our goal is to observe and measure the extent of distinction between the failed and passed executions and study the behavior of this distinction under different fault-inclusions and across programs.

Another aspect of our work is in employing a statistical experimental design. Software quality assurance studies incorporating statistical experimental design are relatively few. Those include several studies focused on test case prioritization, including [8], [1], [11], and on fault localization [20], [7]. These studies had different goals and none of them studied faults as a factor.

III. Notation and Distinction Metrics

The main goal of this paper is to quantitatively explore the behavior of failed executions as reflected in clustering distinction metrics for different programs and fault-inclusions. Techniques which depend on a distinction between failed and passed executions, such as fault localization, failure indexing, execution classification and test case prioritization,

require three components: dynamic execution profile data, a way to compare the profiles, and a method to group or categorize profiles. In [15] techniques that group together failures arising from the same fault (i.e., failure indexing techniques) were each represented as an instantiation of a tuple $< F, D, C >$ – a fingerprinting function F, a distance function D and a clustering function C. Although our study has a different objective, the tuple notation fits well. In this study we instantiate the tuple with function visit count execution profiles for the *fingerprinting function* and Euclidean distance and Hierarchical Agglomerative clustering for the *distance* and *clustering* functions, repectively. It should be noted that our goal is not to explore the best *failure distinction* fingerprinting function, distance metric or clustering method. Rather, we take one of the commonly used fingerprinting functions, distance metrics and clustering methods and study the effect of programs and faults-included on the resulting clustering distinction metrics. However, the general experimental framework proposed in this paper allows any dynamic fingerprinting function, comparison method, and grouping method to be used to study their effects on the distinction metrics presented here.

A. Clustering of Execution Profiles

In our study each faulty program version was run on the Universal Test Suite supplied with that program. In all experiments, whether or not a test case fails was ascertained by differencing the output of the fault-free version with the output of the faulty version. We entered the pass/fail information as well as the control flow events into a database. From the database, we extracted the function level execution profiles for each test case of a faulty version. The $visitcount(VC_{jk})$ of a function k is the number of times it was explicitly called during execution of a test case j. The control flow of a test suite is captured by VC = $[VC_j]$ for $j = 1, ..., N$ test executions.

For each faulty version, we apply clustering to the N test cases of its VC with the object of separating the N_f failing execution profiles from the N_p passing execution profiles. Various clustering algorithms exist with no best method for all situations. We implement Hierarchical Agglomerative clustering which takes a step-wise approach to grouping, where each of N execution profiles is initially assigned to its own cluster. We utilize the often-used Euclidean distance to calculate proximity of any two profiles and merge into one cluster the two that are most similar. The agglomeration can be cut-off at various stages of the merging process. In this study, the clustering is cut-off where the number of clusters equals 30% of the program's test suite size, as in [5], [6].

Once the N test case execution profiles are clustered, each test case belongs to some cluster C_i. We annotate with n_{C_i} the number of test cases in a cluster C_i. Obviously, $n_{C_i} = p_{C_i} + f_{C_i}$ where f_{C_i} and p_{C_i} are respectively the number of failing and passing test cases in a cluster C_i. Clusters can be categorized by fail membership into two types:

- A *fail-pure* cluster (FPC) is one in which all members are failed test cases ($n_{C_i} = f_{C_i}$).
- A *mixed-fails* cluster (MFC) has at least one failed test case ($f_{C_i} > 0$) and one passed test case ($p_{C_i} > 0$).

B. Clustering Distinction Metrics

In order to assess to what extent the execution profiles of failed test cases are different from the execution profiles of passed test cases and to gauge the effect of program and fault-inclusion factors on the results, we introduce four clustering distinction metrics. It should be noted that these metrics are general, as they do not depend on the type or granularity of the profile, nor do they depend on the clustering method.

Fail-Purity (FP) is the percentage of all failed executions that are distinct from any passed executions (i.e., are found in FPCs)

$$\text{FP} = \frac{\sum_{\forall C_i \epsilon \{FPC\}} f_{C_i}}{N_f} \times 100 \qquad (1)$$

Analagous to the *Fail-Purity* metric is the *Pass-Purity* metric, measuring the percentage of all passed executions distinct from any failed executions. However, since our interest is with the failed executions, we focus only on the Fail-Purity metric. To get a sense of this metric's physical meaning, consider the system's unreliability as the portion of all test cases that fail. From a clustering viewpoint, unreliability has two contributions: unreliability due to failures from fail-pure clusters (FPCs) and unreliability due to failures from mixed-fails clusters (MFCs). Thus, $\bar{R}_{system} = \bar{R}_{FPC} + \bar{R}_{MFC}$.

Because the failures found in FPCs show the highest distinction with regard to passes, it is desirable that system unreliability be due mostly to these types of failures. Thus, the FP metric is the *portion* of system unreliability due to failures in FPCs (i.e., FP $= \bar{R}_{FPC}/\bar{R}_{system}$)

Ideally, FP would be 100% meaning that all failed execution profiles are distinct from passed execution profiles. Such distinction may, for example, be useful in fault localization or test case prioritization. When FP is less than 100%, the remaining portion of failed executions are similar to (and clustered with) passed test cases in MFCs In some of these MFCs, a failed execution may be clustered mostly with other failed executions, thus exhibiting a high degree of distinction, or it may be clustered with mostly passes, exhibiting a low degree of distinction. This latter category of MFC clusters, contributing to the MFC portion of unreliability ($\bar{R}_{MFC}$), motivates the following two metrics.

Low-Distinction-Failures (LDF) is a metric focused on the mixed-fails clusters (MFCs). It indicates the portion of all failures that are found in clusters containing a low proportion of failed executions, i.e. these fails are similar to many passed executions. The precise proportion of failures considered a *low* proportion can be user-defined. In this

study, we consider clusters with less than 75% failures (i.e., more than one quartile passes) to exhibit low distinction.

$$\text{LDF} = \sum_{\forall C_i \epsilon MFC | \frac{f_{C_i}}{nC_i} < 0.75} \frac{f_{C_i}}{N_f} \times 100 \quad (2)$$

Low values of LDF are desirable since that would indicate that not many of the failed executions in MFCs exhibit low distinction. The ideal case FP $= 100\%$ corresponds to LDF $= 0\%$. However, even if FP $< 100\%$, it is still possible that LDF $= 0\%$. In that case, the failures not in FPC may be located in clusters with populations of relatively few passes, thus displaying high distinction. From the LDF metric, we learn whether or not MFCs contain predominately failed executions. However, the LDF metric does not reveal how these failed executions are spread across the MFCs.

Mixed-Fails Entropy Potential (H_{MFC}) is a measure of how close to uniform the distribution of failures across the mixed-fails clusters (MFCs) is. The H_{MFC} metric is an indication of the portion of entropy that comes from the dispersion of failed test cases across MFCs in relation to the maximum entropy. Here, the maximum entropy is the most uniform dispersion possible for the failed executions across the mixed clusters. (Measuring the entropy in relation to its maximum enables comparison.) Like the LDF metric, H_{MFC} focuses on the failed executions in MFCs (mixed-fails clusters), collectively annotated N_{f_MFC}.

$$H_{MFC} = \frac{H}{H_{max}} \times 100 \quad (3)$$

where the dispersion entropy of the mixed clusters

$$H = -\sum_{\forall C_i \epsilon MFC} \frac{f_{C_i}}{N_{f\_MFC}} \times \log(\frac{f_{C_i}}{N_{f\_MFC}}) \quad (4)$$

and the maximum dispersion entropy of the mixed clusters

$$H_{max} = -\sum_{\forall C_i \epsilon MFC} \frac{f_{eC_i}}{N_{f\_MFC}} \times \log(\frac{f_{eC_i}}{N_{f\_MFC}}) \quad (5)$$

Note that f_{eC_i} is the number of fails that would be in cluster C_i if the mixed-fails were to be evenly distributed amongst the MFCs. Lower H_{MFC} values are desirable as they indicate less entropy potential; that is, the mixed fails are distributed less evenly and thus there is more distinction. These mixed fails may or may not be clustered with many other passed executions; a viewpoint imparted by LDF.

Average Percent Fails Found (APFF) measures the ability of a test selection method to find early-on the failed executions from among all test executions. This metric is adapted from [8] where it was used to evaluate test prioritization techniques. In our context, APFF is used as a metric to evaluate test case selection methods whose objective is to discover failed executions based upon the clustering. A higher APFF value is desirable, indicating that more of the failed executions were found earlier in the selection process.

APFF reflects the rate of failure discovery as clusters are inspected. Clusters are ordered by some criteria and test cases are selected from these clusters in that order. To calculate APFF, each test case is assigned a priority rank according to the order it is to be inspected. We focus on discovery of the *failed* test cases where FT_i is the rank or ordering position of the i^{th} failing test execution uncovered while examining the ordered clusters.

$$\text{APFF} = 1 - \frac{FT_1 + FT_2 + ... + FT_{N_f}}{N * N_f} + \frac{1}{2N} \quad (6)$$

The equation above finds the area of the curve formed by plotting the fraction of the test suite examined versus the fraction of the failed executions uncovered. A larger area reflects an earlier detection of failures. For details see [8].

In [5], several test selection methods were proposed based upon clustering. Although any of these could be evaluated using the APFF metric as it is a general metric, in this study we focus on the *small clusters* method from [5] which is motivated by their observation that failures are often found in the smallest clusters. Thus far, we have seen that the FP, LDF and H_{MFC} metrics provide views of failure distribution across clusters, but not on the sizes of the clusters into which failures are distributed. Thus, we use this test selection method, Smallest Clusters First (SCF), because it gives another view of failure distinction - whether failed executions are clustered into the smallest clusters.

In the SCF method, test case selection starts by ordering the clusters from smallest to largest such that $\forall j\ nC_j \leq nC_{j+1}$. Each test case within these clusters is assigned an ordering[1]. First, the smallest cluster is identified and all test cases within that cluster are selected and examined for failed executions. Next, all the test cases in the next smallest cluster are selected and examined, and so on. High APFF values for this selection method indicate that failures are either in small fail-pure clusters or are grouped with very few passing executions. This type of distinction can be helpful in observation testing scenarios where there are many test cases to go through and focusing on the failing ones is desirable.

Each of the distinction metrics proposed above measures a different aspect of the distinction between failed and passed executions in clustering. Ideally, high FP, low LDF and H_{MFC} are preferred. For such an ideal case, test selection methods can leverage the distinction and the APFF metric would evaluate these methods favorably. Even with unfavorable values of FP, LDF, and H_{MFC}, for some applications such as observation testing, a test selection method like SCF [5] can still capitalize on distinction as long as MFCs are among the smaller clusters. In such a case, APFF values would be high, indicating early identification of failures and good distinction in regard to method performance.

[1]To simulate the results of averaged repeated random selection from a cluster, failed test cases within a cluster are ordered such that they are selected halfway through the cluster's selection sequence.

IV. Experimental Design

A prime motivation in selecting an experimental design was to study our research questions in such a way as to be able to draw statistically sound observations and conclusions of the effect of factors (independent variables) on the response metrics (dependent variables). We also wanted to avoid confounding factors. Thus, we chose a two-factorial design rather than multiple one-factor designs. In the two-factor design, we control the following factors (i.e., independent variables): *programs* and *fault-inclusions*.

The first step in our experimental work is creating faulty versions of the programs. Each n-fault-inclusion version was created by randomly selecting n faults from a program's set of faults and injecting them into the corresponding program. To help control variability and address the specificity of faults each experiment (program & fault-level combination) was repeated by constructing other faulty versions of that program containing an equivalent number of randomly selected faults. In order to capture the execution profiles for the fingerprinting function, we instrumented the source code of each version using an in-house event-based profiler [25].

Another factor that is important to consider is the *test suite*. The test inputs that the faulty versions are run on are specific to each program. Since our study focuses on the failed executions, creating subset test suites by varying the test cases included (which may or may not trigger the injected faults) would introduce confounding effects. Thus, it is important to note that the test suite is a factor we control by keeping it constant; it is not a factor that we vary and study. Therefore, in order to minimize the influence of the test suite, we utilize the superset of inputs (Universal Test Suite) which does not exclude any fault injected in our versions from manifesting into a failure, thus ensuring that experimental units have the same common denominator.

In selecting an experimental structure that supports our investigation, we considered the inherent attributes of our factors. With a cross-design experiment, one would be able to draw conclusions regarding the interaction effects of the considered factors on the response variables. However, a cross-design requires that each factor be applied equivalently across each level of the other factors [17], which does not apply to our study because faults are specific to the program they inhabit. Faults have differing syntax, context, location and dependencies in program code. Therefore, since the faulty versions created for each fault-inclusion level can only be injected with faults from the corresponding program, the design requires the *fault-inclusion* factor to be nested within each level of *program* factor (Figure 1). For brevity, the figure specifies two programs (P1 and P2) for the *program* factor; in fact, any number of programs can be included in the experiment. Note that there are several repeat observations for each n-fault-inclusion level. For example, there are j repeat observations for program P1 with *fault-inclusion* level *1-Fault*: $P1_{1F_1}$ to $P1_{1F_j}$. Each observation within a fault-inclusion level corresponds to a faulty version which has the same number of faults as other versions within that level, but the particular faults injected into each of these faulty versions vary and are selected at random from those available to that particular program.

Program					
Program 1 (P1)			Program 2 (P2)		
Fault-Inclusion Levels			Fault-Inclusion Levels		
1-Fault	...	m-Faults	1-Fault	...	n-Faults
$P1_{1F_1}$	...	$P1_{mF_1}$	$P2_{1F_1}$	...	$P2_{nF_1}$
⋮	⋮	⋮	⋮	⋮	⋮
$P1_{1F_j}$	...	$P1_{mF_k}$	$P2_{1F_x}$	...	$P2_{nF_y}$

Figure 1. Nested Design

V. Description of the Empirical Objects

Our study employs two empirical approaches, experimentation and case studies, and involves four programs of varying size and complexity: printTokens2 (http://www-static.cc.gatech.edu/aristotle/Tools/subjects/), space (http://sir.unl.edu/portal/index.html), Indent (http://www.gnu.org/software/indent/), and GCC (http://gcc.gnu.org/). Details on the programs used in this paper are summarized in Table I. (All programs were run on a 3.06 GHz Intel Pentium4 with 2 GB RAM and running Ubuntu 8.04.1.)

Table I. Objects in Empirical Study

Program	∼LOC	# Files	#Funcs	# Tests	# Faults
printTokens2	402	3	18	4,054	9
space	6,200	1	157	10,000	23
Indent 2.2.0	10,000	9	56	155	∼30
GCC 3.2.3	300,000	108	2506	2,417	∼151

A. Experimental Objects

For the experimental part of our study we used two publicly available C programs, printTokens2 (pT2) and space, which have been widely used in fault-localization studies, [8], [12], [1], [23] among many others. The lexical analyzer pT2 originated from Siemens Corporate Research and has been made available as part of the Siemens suite. It has a fault-free version as well as ten faulty versions, each containing exactly one injected fault. These faults were inserted by the Siemens researchers in an effort to mimic real-world faults. Nine of the faulty versions involve one line of faulty code and one version involves four faulty lines of code. Two of the faults cannot appear together in the same version. A test suite of 4,115 test cases is also provided. On our platform, some of these test cases produced no execution data; thus we used 4,054 test cases. Also, we discarded pT2 version 10 due to segmentation faults.

The *space* program is an interpreter for an array definition language and was developed by the European Space Agency. It has a fault-free version and 38 faulty versions, each containing exactly one injected fault. These faults are real in that each was previously discovered as an actual fault of the space program. The researchers who originally experimented with space constructed a test suite of 10,000 randomly generated test cases [22], which we used on our experiments.

Most of the faults for space are located on one line of code with a few involving up to two functions and four lines of code. Six of the faulty versions were discared due to segmentation faults. In addition, six of the faulty versions were semantically equivalent to the oracle (i.e., had no failed test cases). Furthermore, for three versions the majority of test cases failed which are not realistic scenarios. Thus, the remaining 23 faulty versions were used for our experiments.

B. Case Studies

Ideally, real-world, large scale programs would make great objects for experimentation. However, experimenting with large programs often is very costly and may not even be possible due to lack of information (e.g., if fault locations are not known one cannot run experiments that require fault injection). In those cases, using *case studies* can provide useful observations and reinforce experimental results. Hence, we include two open source programs as case studies: the medium-sized program Indent, used to beautify C code, and the much larger GCC C compiler.

For these two case studies, we used an analysis method similar to [9]. Indent and GCC each has available a Regression Test Suite run by a script which also acts as an oracle, providing pass/fail label for each test case. Regression test suites often contain test cases to test previous version failures which have been addressed and corrected in the later version. To allow a larger number of failures to be observed, we executed a later version of the regression test suite on an earlier version of the software. Care was taken to eliminate the tests failing due to features introduced in later versions.

GCC is much larger than any of the programs in this study and Indent and space are similar in size. Both Indent and GCC have fault models that are considerably more complex than any faulty version of pT2 and space. In earlier work [9] we identified changes made to fix faults that led to many of the failures (i.e., 27 out of 30 for Indent and 85 out of 151 for GCC). For Indent, 24 failures led to fixes in one file and three were caused by faults spanning two files. As expected, GCC's fault model is even more complex: 49 failures were due to faults spanning multiple files (with some up to 14 files). It should be noted that even faults within a single file usually were affecting much more than a single line of code.

VI. Analysis of Distinction Metrics

In this section we analyze the nested design shown in Figure 1 and present the results. Using analysis of variance (ANOVA) one can ascertain the influence, upon the metrics, of the *programs* factor and also the influence of the *fault-inclusions* factor nested within the *programs* factor. Furthermore, in cases where differences exit, the analysis results answer the question of where the majority of the variability in the distinction measures come from: the programs or the various fault-inclusions within programs.

ANOVA analysis is a parametric method based on the assumptions of normal populations and equal population variances. Moreover, the experimental design is preferred to be balanced since it lends to more robust analysis of variance. The experiment is balanced if it has (i) the same fault-inclusion levels applied to each program and (ii) the same number of repetitions carried-out for each program/fault-inclusion combination. A balanced design, however, limits the number of fault-inclusion levels and repeat observations to the number of versions possible for the program with the fewest number of available faults. This limitation is significant because sometimes a wider range of factor levels may be necessary for an effect to be seen. Thus, in order to leverage the information available, we needed an analysis method robust to unbalanced designs. The nonparametric variance analysis method for two-way nested designs proposed in [21] makes no assumptions on normality or homogeneity of population distributions, nor does it make assumptions on the heteroscedasticity of population variances. Furthermore, since this analysis method allows for an unbalanced experimental design, the number and specific values of levels, as well as the number of repeat observations can vary (as shown in Figure 1). We implemented the nonparametric method presented in [21] using the R statistics package. The Box-adjusted Wald-type statistic utilized in the analysis follows an F distribution which is used to determine the following nonparametric null hypotheses:

- H_0^P: There is no difference in the distributions of the distinction metric (either FP, LDF, H_{MFC} or APFF) among any of the subject programs, and
- $H_0^{F|P}$: There is no difference in the distributions (either FP, LDF, H_{MFC} or APFF) of the distinction measure among the fault inclusion levels within programs.

These nonparemetric hypotheses are stronger than their parametric counterparts as they imply the parametric hypotheses.

In the unbalanced two-factor experiment

- Fault-inclusion levels spanned the range of faults available for each program: 1, 2, 5, 6, 7 faults for pT2 and 1, 2, 5, 6, 10, 15, 20 faults for space.
- Experimental runs per program/fault-inclusion combination were repeated with from at least nine up to 23 faulty versions.

The analysis of variance results for the two-factor experiment are presented in Table II. We first address RQ1 related to the effect of the varying programs on distinction metrics. For each of the distinction metrics, variance in the respective distributions between programs is statistically significant ($> 99.4\%$ confidence interval). That is, the results yield enough evidence to reject the null hypothesis H_0^P in favor of the alternative hypothesis H_a^P that there is a difference in the distinction metric distributions across programs. The program factor is responsible for contributing the majority of the variance. For the LDF and H_{MFC} metrics, 62% and 59% of variability, respectively, is due to the programs. Even more notable are the FP and APFF metrics, with over 98%

of their variability coming from the programs.

Table II. ANALYSIS OF VARIANCE FOR PROGRAMS (P) VS NESTED FAULT-INCLUSIONS ($F|P$)

Distinction Metric	Factor	Wald-type Box-Adjusted Rank Statistic	p-value	H_0	Cont to Var
Fail Purity (FP)	P	153.2691455	1.15E-17	Rej	98.20%
	$F\|P$	2.808993457	0.017019	Rej	1.80%
Low Distinction Fails (LDF)	P	9.261079462	0.003393	Rej	62.78%
	$F\|P$	5.489803863	4.53E-05	Rej	37.22%
Mixed-Fails Entropy Pot (H_{MFC})	P	8.144746239	0.005598	Rej	59.39%
	$F\|P$	5.570347305	8.74E-06	Rej	40.61%
Average % Fails Found (APFF)	P	327.4113655	7.38E-27	Rej	99.50%
	$F\|P$	1.634288668	0.150367	DNR	0.50%

To address RQ2 and RQ3 we rely on Figure 2 which shows the distinction metrics resulting from the two-factor experiment and the two case studies. For pT2 and space the boxplot summarizes the distinction metric restults for each program/fault-inclusion combination. (The x-axis shows the number of fault-inclusions and the y-axis shows the corresponding distinction metric result.) For the case studies Indent and GCC at the right end of Figure 2 we show the corresponding distinction metrics for one release of each respective program. Both Indent and GCC have more faults than pT2 and space (i.e., approximately 30 faults for Indent and 151 faults for GCC). Next we take a closer look at the results for each distinction metric.

Fail-Purity (FP) FP is the contribution to unreliability due to the failures in the Fail Pure Clusters. Practically, to aid the testing process, it is better to observe higher FP values (i.e., more failing execution profiles are dissimilar from any passing execution profiles). The FP boxplots in Figure 2 show overall higher FP values, and thus higher distinction, for pT2 than for space. The effect of the fault-inclusion levels on FP, although less prominent, is also statistically significant (see Table II). However, pT2 and space show different trends as fault-inclusion levels increase. While FP for pT2 shows a decreasing trend with diminishing distinction as fault-inclusions increase, for the space program FP tends to eventually increase with increasing fault-inclusions. We see that case studies Indent and GCC, which have higher number of more complex faults, have low FP values. Fail-purity for GCC is from the lowest values observed.

Low Distinction Fails (LDF) is the part of unreliability due to MFCs (clusters containing both failed and passed test cases), specifically the clusters containing less than 75% failures. For the LDF metric, distinction is better with lower values. Overall, pT2 tends to have lower LDF values (i.e., higher distinction) than space. The differences in LDF values for increasing fault-inclusion levels for each program are statistically significant (see Table II). And similar to distinction results reflected by FP, the LDF values tend to increase for pT2 (decreasing distinction) and decrease for space (increasing distinction). The case studies display much less distinction than the experimental programs. The LDF value for Indent is much higher than for pT2 and approximately the same as the higher range of space values.

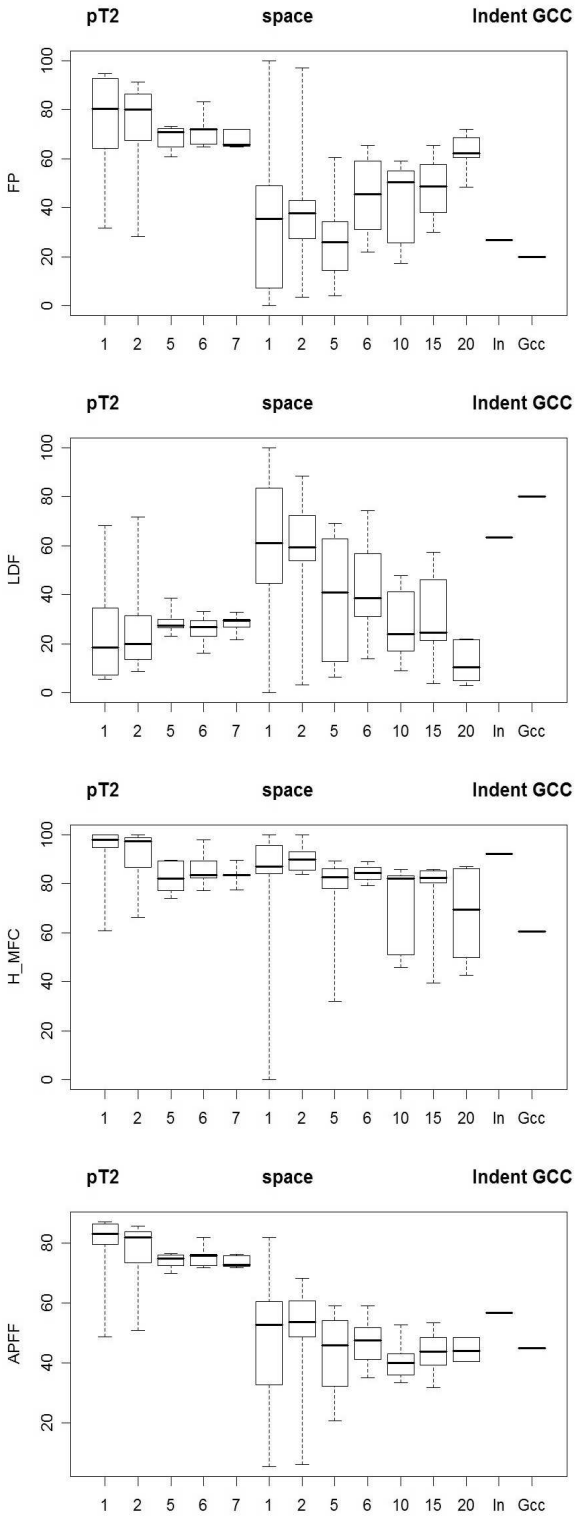

Figure 2. Distinction Metrics: FP, LDF, H_{MFC}, APFF (x-axis shows the number of fault-inclusions for each program)

The GCC LDF value was from the highest observed.

Mixed-Fails Entropy Potential (H_{MFC}) gives an indication of the entropy that comes from the dispersion of failed test cases in MFCs only. Lower values indicate less entropy and thus, less uniform dispersion of failed executions among the mixed clusters. H_{MFC} values for pT2 are slightly higher than those for space, with entropy potential tending to decrease for each program as fault-inclusion levels increase. The differences due to the effect of fault-inclusions per program are also statistically significant (see Table II), contributing 40% to the variance. The case study Indent has slightly higher entropy potential than multi-fault versions of pT2 and space. GCC, however, has lower H_{MFC} than either pT2 or space. The following observations are interesting to note. The values of H_{MFC} are rather high (i.e., typically higher than 50%) which indicates significant (approaching uniform) dispersion of failing executions among MFC clusters. However, H_{MFC} decreases with increasing levels of fault-inclusions, which means that as we observe more failures their execution profiles tend to be similar and group in larger numbers together.

Average Percent Fails Found (APFF) The APFF metric reflects the rate failures are found as test cases are selected starting from the smallest clusters. Higher values indicate more failed test cases are found sooner in the selection process. The APFF values are higher for pT2 than for space, indicating that failed executions tend to cluster in smaller clusters more for pT2 than for space. Visually, the boxplots in Figure 2 show that APFF values for pT2 and space tend to decrease slightly and converge as the fault-inclusion levels increase. However, the differences due to the effect of the fault-inclusions per program appear not to be statistically significant. The case studies have lower APFF values than pT2 and similar values to those for space.

Summary of the results. With regard to RQ1, we conclude that there is a statistically significant difference of distinction metrics between programs, with programs contributing from 59% to as much as 99.48% to the variability. In general, the metric values for pT2 are more distinct than those for space, Indent and GCC (the notable exception is the entropy metric). That is, the differences in executions appear to manifest more and are captured better by the control flow visit counts for pT2 than for space, Indent or GCC.

For RQ2 we seek to find if fault-inclusions affect the distinction metrics. Indeed, it appears that distinction metrics are affected as the fault-inclusion levels increase. The effect of the fault-inclusions per program is statistically significant for all distinction metrics except APFF. In general, fault-inclusions per program contribute much less to the variability of the differences than do the programs.

With respect to RQ3 the results show that the distinction metrics do not always follow the same trend as the fault-inclusions levels per program increase. Specifically, it is clear that the trends of the FP and LDF distinction metrics differ among programs; for pT2 the distinction decreases, whereas for space it increases. On the other hand, for the H_{MFC} and APFF metrics, the trends with pT2 and space are similar; both show an increasing distinction trend for H_{MFC} and a converging, slightly decreasing trend for APFF.

Taking all the metrics together for each program helps further explore RQ3. Let us first focus on pT2 occupying the leftmost column of Figure 2 boxplots. This program shows high distinction for all but the entropy metric. We can interpret that, as the fault-inclusions increase, the portion of unreliability due to FPCs decreases (i.e., we observe decreased FP), which results in a greater portion of failed executions clustered in mixed clusters. These failing executions are apparently dispersed into larger clusters (i.e., APFF values decrease) containing more passes (i.e., LDF increases). However, as the fault-inclusions increase more of these failing executions gather together in the same clusters (i.e., H_{MFC} decreases).

Intuitively, one may expect to observe the same decreasing distinction for space as for pT2 with the increasing fault-inclusion levels. However, looking at space's boxplots shown in the middle column of Figure 2, we observe that the FP, LDF, and H_{MFC} metrics for space show increasing distinction as fault-inclusion levels increase. Maybe this is not surprising if we consider that with increasing fault-inclusions, we usually observe more failures. If these failed executions have similar profiles they will tend to cluster together, which will lead to more Failure Pure Clusters (i.e., higher FP) and/or more failed executions clustered together in mixed clusters. The more failed executions cluster together, the lower will be the LDF and entropy measures.

What prompts this difference in behavior between pT2 and space? Examination of pT2's failure behavior reveals one possible explanation. As was reported in [4] for all the Siemens programs collectively, we found that multi-fault combinations of pT2 cause failures that can exhibit any of independence, destructive and/or constructive interference. On the other hand, space's multi-fault combinations largely exhibit independence in the failures manifested. It appears that the more complex fault interactions of pT2 may result in more different execution profiles and thus result in decreasing distinction.

As for the case studies Indent and GCC (the two points in the right-most column of Figure 2), the distinction metric results mostly indicate that these high fault-inclusion programs exhibit low distinction of failing executions. The notable exception is GCC's lower entropy potential H_{MFC} value, which is on the low side of values for space's 20-fault versions. It appears that although GCC has low fail purity, the large number of its failed executions in mixed clusters are clustered together.

VII. Threats to Validity

Our empirical study is subject to threats to validity which we discuss in terms of construct, internal, conclusion, and external threats. In case of *construct validity* we are

concerned with ensuring that we are actually testing in practice what we meant to test. A potential threat is in having an empirical study with confounding factors. We address this threat by conducting a two-factor experiment that studies the effects of both programs and faut-inclusions. Furthermore, the specificity of faults to their corresponding programs is addressed by nesting the fault-inclusion levels within programs. Since we do not have control over the types of faults corresponding to a program, we select randomly the faults to inject in the faulty versions of each program, and perform repeat runs for each program/fault-inclusion combination. Another potential threat to construct validity is related to choosing a sufficient number of levels of the fault-inclusion factor. Sometimes just the fact that several levels of a factor are being investigated may not be enough; the number of levels utilized and the actual values of the levels used may be instrumental in determining if an effect is demonstrated. To address this threat to validity we opted for an unbalanced design which allowed us to fully utilize the greater number of faults available for the space program. Furthermore, we were careful to include most of the same levels for both programs as well as spread out the levels selected for space in order to avoid a multiplicative effect on the results. The final threat to construct validity is related to the mono-method bias, i.e., to the use of a single metric to make observations. We addressed this threat by introducing four distinction metrics, each measuring different aspects of the groupings of failing and passing executions.

Threats to *internal validity* arise when there are sources of influence that can affect the independent variables and/or measurements. One potential source of influence is the instrumentation of the code which can affect the results if not done accurately. To ensure accuracy in collecting function level events, we implemented an event-based profiler, rather than use a sampling-based one. Another potential threat to internal validity that can influence the results is the types of injected faults, which is governed by the availability of faults for each of the experimental object programs. We partially addressed this threat by randomly selecting the fault(s) to inject in multi-fault versions, and for each cell in our experimental design ran repetitions with different combinations of faults. Including two case studies which have significantly more complex faults causing failures, additionally addresses this threat to validity. Another potential source of influence is the test suites used in the experiments for testing the faulty versions. By using each program's 'universal' test suite, we ensure the following: (1) there are test cases that exercise the faults injected in each faulty version and (2) variations in the response variables (i.e., distinction metrics) which could be due to the choice of test cases are eliminated.

In considering threats to *conclusion validity*, we examine the ability to draw correct conclusions (i.e, statistical validity). We addressed this threat by utilizing a formal statistical experimental design, as well as conducting repeat observations for each cell (i.e., program/fault-inclusion combination). Furthermore, we carefully checked our data for validity of the assumptions made by the analysis methods and applied an appropriate nonparametric statistical method, which is also robust to unbalanced design.

External validity considerations determine to what extent results can generalize. We performed our experiment on programs widely used for empirical studies in related fields. One program, pT2, is small and has faults seeded by humans, while the other program, space, is medium-sized with real faults detected during testing and operational usage. Moreover, we extend the analysis to case studies using two open source programs whose faults are real and more complex. Although the case studies do not include replication (results are based on one release per case study) and thus are not included in the statistical analysis, they help demonstrate the trends and add further confidence to the observations.

VIII. Conclusions

We have presented an empirical study aimed at exploring the distinction between the dynamic profiles of failing and passing software executions. Specifically, we focused on the control flow profiles at function level as they have been used widely in work related to fault localization, failure indexing, execution classification, and test case prioritization. Yet, none of the related work explored and quantified the extent of distinction between failing and passing executions and the effect that different factors have on the distinction. We used clustering as a natural way to group software execution profiles and proposed a set of distinction metrics which measure different aspects of the results of the groupings.

A major contribution of this work is the experimental framework based on a well-defined experimental design, which is not restricted only to the type and granularity of the execution profile, clustering technique, and response variables used in this paper. It allows for incorporating additional empirical objects and, more importantly, investigating other fingerprinting functions, distances and clustering methods, and other response variables. The quantification of the contribution different factors may have on the distinction of the failing and passing executions underpins the better understanding and utilization of dynamic fingerprinting functions for automated verification and validation, fault localization and other software quality assurance studies.

Another contribution of this work is related to the empirical observations made based on the experimental objects and case studies. We see statistical evidence that the program with its inherent characteristics contributes more significantly to the variation in the clustering distinction metrics than do the fault-inclusion levels. Thus, when testing is at a phase where there are still a significant number of faults present, the accuracy/feasability of techniques for automatic identification of failed test cases and fault localization may be influenced significantly by the program under study. Furthermore, for a given program, our results show that

increasing fault-inclusion levels does not always lead to worse distinction between failing and passing executions.

We conclude with a brief description of the implications of our work. (1) This paper illustrates the necessity of conducting controlled multi-factor experiments which follow the general principles of random selection and replication for each cell of the experimental design. (2) The variability among programs as reflected by the distinction metrics emphasizes the fact that empirical studies must be conducted on a variety of programs in order to be able to gauge effectiveness, adjust approaches and methods, and ensure external validity. (3) Fault-inclusion levels with multi-fault versions need to be part of empirical studies since they typically have statistically significant effect on distinction metrics. How much influence they exert and what the trend of that influence is seem to depend on the program and the type of faults inherent to that program. (4) Variability in distinction metrics decreases as the levels of fault-inclusion increase. Therefore, it is important to use different numbers of faults in experiments as well as a variety of combinations of faults. (5) Experiments allow for controlled manipulation of factors (i.e., independent variables), replication, and testing the statistical significance of the results, which promotes sound and generalizable results. However, since formal experiments must be carefully controlled, by their nature they are "research-in-the-small", which imposes risk when one attempts to apply methods developed in a laboratory to a real project. To avoid scale-up problems, provide a realistic view and give check to experimental results, we advocate the use of case studies based on typical industrial scale programs (i.e., "research-in-the-typical") in combination with controlled experiments.

IX. Acknowledgments

This work was funded in part by the NSF under the grants CNS-0447715 and CCF-0916284.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th Int'l Conf. Softw. Eng.*, pages 402–411, 2005.

[2] J. Bowring, J. Rehg, and M. Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Softw. Eng. Notes*, 29(4):195–205, 2004.

[3] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *31st Int'l Conf on Softw Eng (ICSE 2009)*, Vancouver, Can, May 2009. ACM SIGSOFT and IEEE.

[4] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. *Int'l Symp. Softw. Reliability Eng*, pages 165–174, 2009.

[5] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *23rd Int'l Conf. Soft. Eng.*, pages 339–348, 2001.

[6] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *8th European Softw. Eng. Conf.*, pages 246–255, 2001.

[7] S. Elbaum, S. Kanduri, and A. Andrews. Trace anomalies as precursors of field failures: an empirical study. *Empirical Softw. Eng.*, 12(5):447–469, 2007.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[9] K. Goševa-Popstojanova, M. Hamill, and X. Wang. Adequacy, accuracy, scalability, and uncertainty of architecture-based software reliability: Lessons learned from large empirical case studies. In *17th Int'l Symp on Softw. Reliability Eng.*, pages 197–203, 2006.

[10] M. Haran, A. Karr, M. Last, A. Sanil, A. Orso, A. Porter, and S. Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Trans on Softw Eng*, 33(5):287–304, 2007.

[11] D. Hyunsook and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. on Softw. Eng.*, 32(9):733–752, 2006.

[12] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th Int'l Conf. on Autom. Softw. Eng.*, pages 273–282, 2005.

[13] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *24th Int'l Conf. on Softw. Eng.*, pages 467–477, 2002.

[14] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *27th Int'l Conf. Softw. Eng.*, pages 412–421, 2005.

[15] C. Liu, X. Zhang, and J. Han. A systematic study of failure proximity. *IEEE Trans. Softw. Eng.*, 34(6):826–843, 2008.

[16] W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Trans. on Softw. Eng.*, 33:454–477, July 2007.

[17] D. Montgomery. *Design and Analysis of Experiments*. J Wiley and Sons, New York, NY, 2001.

[18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *25th Int'l Conf. Softw. Eng.*, pages 465–475, 2003.

[19] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *Int'l Conf. Autom. Softw. Eng.*, page 30, 2003.

[20] J. R. Ruthruff, M. Burnett, and G. Rothermel. Interactive fault localization techniques in a spreadsheet environment. *IEEE Trans. on Softw. Eng.*, 32:213–239, 2006.

[21] A. Stavropoulos and C. Caroni. Rank test statistics for unbalanced nested designs. *Statistical Methodology*, 5(2):93 – 105, 2008.

[22] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Int'l Conf. Softw. Maintenance*, page 44, 1998.

[23] W. Wong, Y., L. Zhao, and K. Cai. Effective fault localization using code coverage. In *31st COMPSAC*, pages 449–456, 2007.

[24] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *ICSE '08: Proc of 30th int'l conf on Softw. eng.*, pages 201–210. ACM, 2008.

[25] J. Zemerick. Profiling, extracting, and analyzing dynamic metrics. Master's thesis, West Virginia University, 2008.