AMERICAN UNIVERSITY OF BEIRUT

MINING FOR SIGNIFICANT EXECUTION PROFILES FOR SOFTWARE ASSESSMENT

by
JOAN MOUNIR FARJO

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
May, 2014

AMERICAN UNIVERSITY OF BEIRUT

MINING FOR SIGNIFICANT EXECUTION PROFILES FOR SOFTWARE ASSESSMENT

by
JOAN MOUNIR FARJO

Approved by:

[illegible]

Prof. Wassim Masri,Associate Professor — Advisor
Electrical and Computer Engineering

[illegible]

Prof. Hazem Hajj, Associate Professor — Member of Committee
Electrical and Computer Engineering

[illegible]

Prof. Fadi Zaraket, Assistant Professor — Member of Committee
Electrical and Computer Engineering

Date of thesis defense: April 22, 2014

ACKNOWLEDGMENTS

Although this is considered to be the first page in the thesis, but I am just writing it at the end. To admit, it is the page that is taking the longest time to complete, since I want to thank every person who helped me throughout this journey.

First and foremost, I would like to thank my advisor Dr. Wassim Masri for his advices all throughout my thesis work and for his support and complete understanding of what I have been passing through from stress of work and other factors that made my journey quite cumbersome. As I would like to thank Dr. Fadi Zaraket and Dr. Hazem Hajj for being part of my thesis committee.

Many thanks go to Rawad who was always there when I wanted to share with him my ideas and my assumptions to get some valuable feedbacks. Although some of the feedbacks were shocking and I had to reevaluate my assumptions, but all turned out to be helpful in the end.

To Pierre, my “Ladoudzi”, whom I admit that he shared with me every single phase of my thesis. While carpooling and being stuck in traffic for almost an hour, he had to listen to my research updates every single morning. He shared with me my worries and continuously asked about my results. Thank you for the times you had to wake up quite early so I could catch a class or a meeting, thank you for your “interpolations” when I used to wait for long simulations to end.

To Nadida, my “Twinni”, who was always there supporting me. Thank you for the numerous times I had to sleep over at your place after long hours of work, thank you for never getting annoyed when I had to continuously bother you with the server thingy. And most of all, thank you for the times when I really needed a break and you were there for a cup of coffee or a “delicious” dinner at Little China.

To Nayla, Rana, and Lara who were always ready for some times of relaxation and for their random visits to cheer me up. I would rather prefer to remember the fun time we spent during our semester breaks, rather than the times we spent at each other`s place working on our projects and me preparing the famous “soup Maggi” that you all hate.

Thank you to Mario for the online chat entertainment late at night while I was working and he was watching one series after the other. Thank you for showing up on Saturdays and Sundays and going to eat crazy fatty stuff. Thank you to Naoum who was asking about my progress almost EVERY SINGLE day. Thank you to all my friends whom I got to know from AUB, thank you to Veloxent for their continuous understanding, thank you to all my colleagues, and to all my dear JM friends.

Well friends were part of my journey, but my family is the biggest part of my journey. They were and will always be my endless support in my life. Thank you Manoura, Mimiza, Couci, and Janjoun for every single day you had to suffer from my swinging moods and for your worries while I stayed working late at night. Thank you for continuously preparing the cupS of Nescafe. But sorry for the numerous times I did not spend with you, and will definitely make up for this as I will promise you, or I will try to promise you, that I won`t pursue any additional degree. I know we have passed by the toughest time ever and because of all five of us we are able to stand up and continue our journey of life, which I only hope it will bring us just health and happiness.

I want to dedicate a special thank you to Manoura and Mimiza, who always believe in me, trust every single move I do, and are proud of me.

Last but not Least, I cannot but thank God for almost everything. Thank you God for this family and these friends. Thank you God for the energy you always push in me and for the patience you always give me. Thank you for the tough moments and the joyful ones!

AN ABSTRACT OF THE THESIS OF

Joan Mounir Farjo for Master of Engineering
Major: Electrical and Computer Engineering

Title: Mining for Significant Execution Profiles for Software Assessment

The interest in applying data mining and statistical techniques to solve software analysis problems has increased tremendously in recent years. Researchers have presented numerous techniques that mine and analyze execution profiles to assist software testing, fault localization, and program comprehension.

Previous empirical studies have shown that the effectiveness of such techniques is likely to be impacted by the type of the profiled program elements. This work further studies the impact of the characteristics of execution profiles by focusing on their size; noting that a typical profile comprises a large number of elements, in the order of thousands or higher. Specifically, we devised six reduction techniques and comparatively evaluated them by measuring the following: 1) reduction rate; 2) information loss; 3) impact on the quality of cluster analysis, using various metrics; 4) cost of reduction; and 5) impact on two software analysis techniques, namely, cluster-based test suite minimization and profile-based online intrusion detection.

Our results were promising as: a) the average reduction rate ranged from 92% to 98%; b) three techniques were lossless and three were slightly lossy; c) the quality of cluster analysis was not deteriorated; d) the cost of reduction was not very significant; and e) reducing execution profiles noticeably benefited the two software analysis techniques in our experiments.

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER I

INTRODUCTION

In recent years, researchers in the dynamic program analysis field have extensively used data mining and statistical techniques to address various problems. The focus was mainly on mining and analyzing execution profiles to assist software testing, fault localization, and program comprehension.

All throughout the chapters; execution profiles, profiling elements, and features will be used interchangeably.

Since execution profiles are essential to the developed techniques, their characteristics needed to be studied. Previous empirical studies have shown that the type of the profiled program elements is likely to impact the effectiveness of the techniques [1][23][34].Hereafter, program elements represent statements, branches, def-uses, information flow pairs [30], slice pairs [30], paths [3], and possibly other program constructs that are also of structural nature. In this work, we are concerned with the size of execution profiles; noting that a typical profile comprises a large number of program elements, in the order of thousands or higher. According to the curse of dimensionality, an increase in the size of the execution profiles might diminish the effectiveness of the techniques at hand. Execution profiles comprising a large number of program elements (or variables) represent high-dimensional data. Given that a higher dimensionality results in a larger volume of the data space, the data points become sparse, which has a negative effect on data mining and statistical techniques. For example, cluster analysis aims at finding groups of data points that are similar, but if all data points are sparse, the analysis is likely to fail even in the presence of truly similar points. Therefore, reducing the size of execution profiles is desirable as a preprocessing step to clustering or any other profile-based technique. As a side note, structural execution profiles are expected to contain a lot of redundancy mainly due to the transitivity relationships induced by control and data dependences [22], which suggests that high levels of reduction might be achieved.

A well-established approach that we previously used [8] to reduce the high dimensionality and redundancy in execution profiles is Principal Component Analysis (PCA) [11][42]. But PCA transforms the original data to a new coordinate system, which is problematic for many software analyses since this might negatively impact the interpretability of learning models and the extraction of useful intrinsic properties.

The following motivating example demonstrates how a set of execution profiles comprising 7 features could be represented using only 2 features without any loss of information. Recall that: a) an execution profile is induced by the execution of a given test case; and b) a feature represents the execution of some program element such as a statement, a branch, or a def-use pair. Table 1 shows the values of the features {f_1, f_2, f_3, f_4, f_5, f_6, f_7} for each of the execution profiles {t_1, t_2, t_3, t_4, t_5}. For example, in the first row, t_1 indicates that the features f_1, f_3, f_4, f_6, and f_7 did execute, whereas f_2 and f_5 did not. Table 1 represents what we will refer to as *execution matrix*, hereafter.

Table 1. Original execution profiles

	f1	f2	f3	f4	f5	f6	f7
t_1	1	0	1	1	0	1	1
t_2	0	1	0	1	1	1	0
t_3	0	1	0	0	1	0	0

t_4	0	1	0	1	1	1	0
t_5	1	0	1	0	1	0	1

It can be observed that f_1, f_3, and f_7 exhibit the same pattern of occurrence in all five profiles; and similarly for f_4 and f_6. Therefore, {f_1, f_3, f_7} could be replaced by f_1, and {f_4, f_6} by f_4, for example. This yields reduced execution profiles comprising only 4 elements, namely, {f_1, f_2, f_4, f_5}, shown in Table 2. We refer to this type of reduction as *Basic Redundancy Removal.*

Table 2. Reduced execution profiles

	f1	f2	f4	f5
t_1	1	0	1	0
t_2	0	1	1	1
t_3	0	1	0	1
t_4	0	1	1	1
t_5	1	0	0	1

However, the resulting profiles still contain some redundancy. Specifically, knowing the execution status of subset {f1, f4} in any profile, we can infer the execution status of the remaining program elements. Table 3 summarizes this relationship. For example, the second row indicates that every test case that exercises f4 but not f1, exercises both of f2 and f5. Following this second observation, the execution profiles in Table 2 can be reduced down to include only two elements, f1 and f4. For completeness, Table 4 shows how {f2, f3, f5, f6, f7} could be inferred from {f1, f4}.

Table 3. Inferred execution profiles

{f1, f4}		{f2, f5}
t_1	11	00
t_2, t_4	01	11
t_3	00	11

t_5	10	01

Table 4. Further inferred execution profiles

{f1,f4}		{f2, f3, f5, f6, f7}
t_1	11	01011
t_2, t_4	01	10110
t_3	00	10100
t_5	10	01101

As demonstrated in the above example, the dimensionality of execution profiles could be reduced considerably while preserving the original coordinate system and without any loss of information.

In this work we focus on reduction techniques that preserve the original coordinate system to evade the limitations that PCA suffers from. In other words, we will be concerned with feature selection techniques [41] as opposed to feature extraction techniques [11]. Specifically, we will investigate selection techniques categorized as: 1) filter models based on the symmetric uncertainty measure [12]; 2) wrapper models based on the genetic algorithm [12]; and 3) hybrid models that combine filter and wrapper models.

We evaluated our techniques using ten subject programs by measuring the following: 1) reduction rate; 2) information loss; 3) impact on the quality of cluster analysis, using various metrics; 4) the cost of running the reduction; and 5) impact on two software analysis techniques, namely, cluster-based test suite minimization (Tech-I) [5][6][34], and profile-based online intrusion and failure detection (Tech-II) [26].

Our reduction techniques exhibited: 1) very high rates of reduction with no to very little loss of information in most cases and insignificant reduction running time cost along with almost no deterioration of the clustering measures; 2) a major positive impact on the effectiveness and efficiency of Tech-I, which relates to the curse of dimensionality; and 3) a major positive impact on the efficiency of Tech-II. This shows that reduction can potentially improve the efficiency and/or effectiveness of program analyses that use execution profiles.

The main contributions of this work are the six techniques for reducing execution profiles, two of which are based on the symmetric uncertainty measure, two on the genetic algorithm, and two combine both approaches; besides an empirical study evaluating our reduction techniques, and yet another empirical study demonstrating the value of reduction in software analysis. The flowchart presented in Figure 1 illustrates the overall work flow.

The rest of the thesis is divided as follows: Chapter 2 surveys related work. Chapter 3 provides reduction techniques overview. Chapter 4 describes the proposed reduction techniques. Chapter 5 presents our empirical study that assesses the quality of reduction. Chapter 6 presents our empirical study that demonstrates the impact of the reduction methods on two software analysis techniques. Chapter 7 presents conclusions and future work.

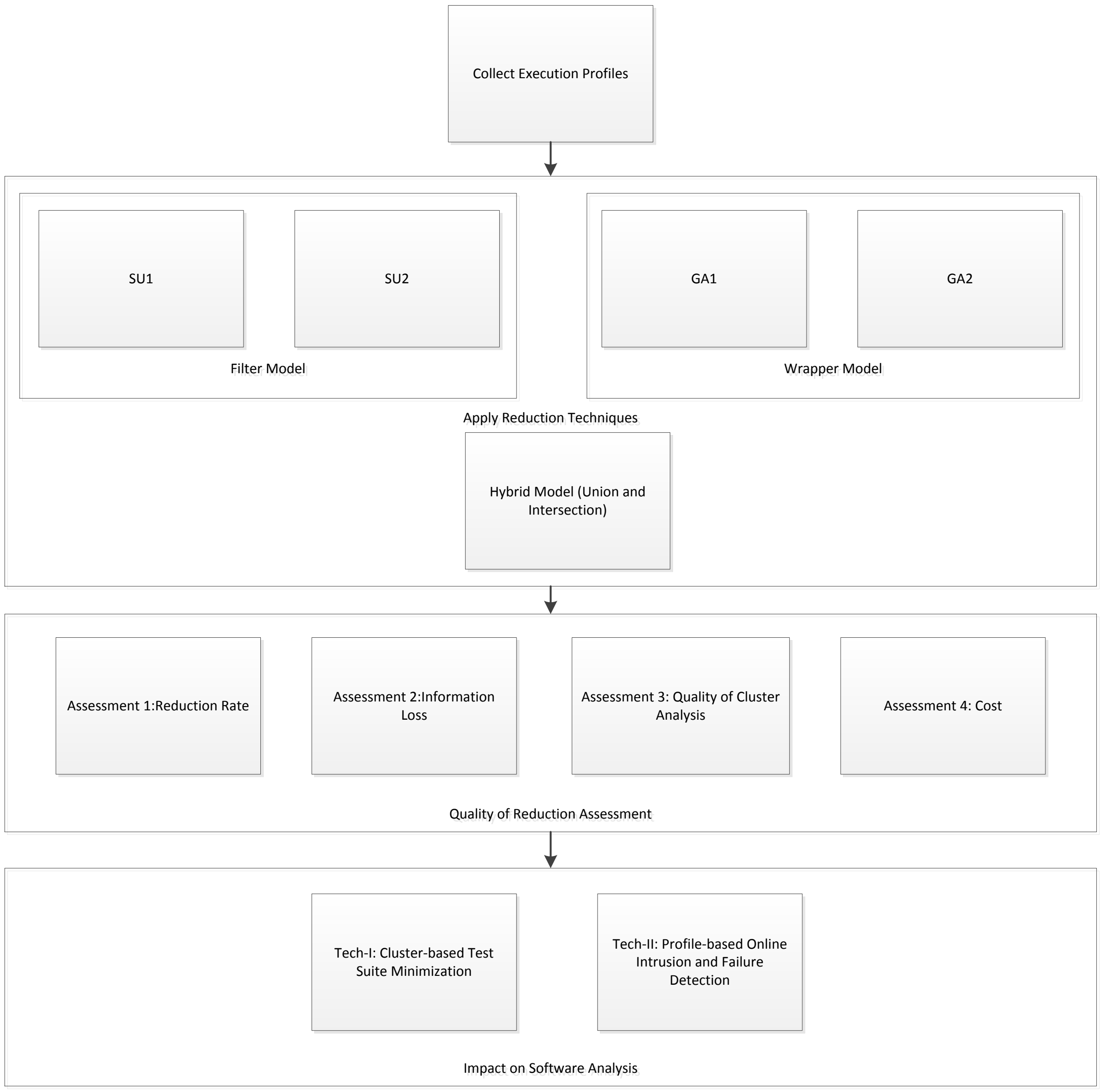

Collect Execution Profiles
SU1
SU2
Filter Model
GA1
GA2
Wrapper Model
Apply Reduction Techniques
Hybrid Model (Union and Intersection)
Assessment 1:Reduction Rate
Assessment 2:Information Loss
Assessment 3: Quality of Cluster Analysis
Assessment 4: Cost
Quality of Reduction Assessment
Tech-I: Cluster-based Test Suite Minimization
Tech-II: Profile-based Online Intrusion and Failure Detection
Impact on Software Analysis

Figure 1. Overall Work Flow

CHAPTER II

RELATED WORK

We are not aware of any body of work that investigates the techniques for the benefits of reducing execution profiles. However, there exist only two attempts related to reducing execution profiles.

The first attempt for reducing execution profiles for the purpose of improving software analysis is described in [8]. In that work, we investigated *Principal Component Analysis* (*PCA*) as a reduction technique and measured its impact on two cluster-based analysis techniques, one aiming at identifying coincidentally correct tests [24][25], and the other at test suite minimization [34]. Our experimental results showed that the impact was positive on the first, but inconclusive on the second. *PCA* reduces the dimensionality of a data set (possibly involving correlated variables) to a new set involving uncorrelated variables. The generated uncorrelated variables are called *principal components* (*PC*s). The obtained set has the *PC*s ordered by the fraction of the total information/variation each retains. That is, the first *PC* captures as much of the variability present in the data set as possible, the second *PC* also captures as much of the variability but under the constraint of being uncorrelated with the previous (first) *PC*, and similarly for the subsequent *PC*s. After applying *PCA*, only the first few *PC*s are retained and the remaining ones ignored. Note that *PCA* transforms the original data to a *new* coordinate system. Therefore, it is not possible to recover the non-redundant profiling elements from the retained *PC*s, which renders many software analysis techniques inapplicable. For example, in coverage based fault localization [1][15][28], identifying failure-correlated *PC*s will *not* lead to the failure-correlated program elements, which are normally needed to locate the fault. This and the fact that our experimental results were inconclusive called for further investigation.

The second attempt for reducing execution profiles is described in [2], in which a lossless technique that uses a genetic algorithm was presented. Its impact was empirically evaluated on the quality of clustering and greedy test suite minimization. The results were very promising as the reduction rate ranged from 94% to 99% with a negligible deterioration in the quality of clustering and greedy minimization. Following [2], a considerable negative impact on cluster-based test suite minimization was observed, which led us to investigate reduction techniques further.

Since very limited work was observed in the software field, we ought to refer to the data mining literature that contains numerous articles relevant to different reduction techniques from which our proposed methods emerged.

CHAPTER III

REDUCTION TECHNIQUE OVERVIEW

Dimensionality reduction techniques are categorized into two major approaches: feature extraction and feature selection. Feature extraction techniques project high dimensionality features onto a new space of lower dimensions; thus leading to a new set of features. This projection can be a linear projection where the newly projected features have a linear combination or a non-linear projection. Since this feature transformation might be problematic for many software analyses, we restrict our investigation to feature selection algorithms.

Feature selection algorithms [41] do not alter the original representation of the features, but choose a subset of them according to a certain selection criterion; thus preserving their semantics. Following the way class label information is used [47], feature selection algorithms are categorized as:

- Supervised, where the feature relevance is determined by evaluating feature's correlation with the class
- Unsupervised, where the search for relevant information is guided without class labels but by exploiting different measures
- Semi-supervised, where small amount of labeled data is used as additional information to improve the performance of unsupervised feature selection

But this categorization is not relevant to our work since we seek reduction techniques that do not require the availability of class labels. Another categorization, which is relevant to our work, is based on the adopted selection strategy. According to the latter, a feature selection algorithm falls into one of three models: filter, wrapper, or hybrid. Next, we present relevant background concerning these models.

A. Filter Model

Filter models examine the intrinsic properties of the data to evaluate the features without involving any learning algorithm [13][44].It is usually cast into a binary selection of features; specifically, given a set of features, the goal of a filter based feature selection is to choose a subset from that set that maximizes some criterion. Thus, the important aspects are the choice of the selection criterion and the selection algorithm that proceeds until a pre-specified number of features are selected or some threshold is met. It is important to note that the filter approach does not rely on searching the space of feature subsets; instead it selects features on the basis of statistical properties. Each feature is ranked according to a statistical property, with the score reflecting the discriminative power of each feature.

Various measures could serve as the filter model; these include but are not limited to the following scores: data variance [16][44], Fisher score [7][13], Laplacian score [16], Relief-F score [19][43], SPEC [46][47], Pearson correlation [44], Chi-Square [20], Gini index [39], and information gain [33].

1. Data Variance

Data variance finds features [16][44] useful for representing data based on the variance along a dimension that reflects its representative power. However, there is no reason to assume that these features are useful for discriminating between data in different classes.

2. Fisher Score

Fisher sore [7][13] tries to find a subset of features, such that the distances between data points in different classes are as large as possible, while the distances between data points in the same class are as small as possible. It assigns the highest score to the features that assign similar values to the samples from the same class and different values to samples from different classes. The top-*k* ranked features are selected based on the largest scores, where the features had been treated independently. The fisher score is denoted as:

$$\mathbf{FS(F_r)} = \frac{\sum_{j=1}^{c} \mathbf{n_j}(\boldsymbol{\mu_{r,j}} - \boldsymbol{\mu_r})^2}{\sum_{j=1}^{c} \mathbf{n_j}(\boldsymbol{\sigma_{r,j}})^2} \qquad (1)$$

where μ_r is the mean of the feature F_r, n_j is the number of samples in the j-th class, $\mu_{r,j}$ and $\sigma^2_{r,j}$ are the mean and variance of F_r on class *j*. It is known that Fisher Score is a special case of Laplacian Score, with a change in the similarity matrix as will be discussed in the next point.

3. Laplacian Score

Laplacian Score evaluates the features according to their locality preserving power, which is modeled by constructing a nearest neighbor graph where the importance of a feature is thought of as the degree it respects the graph structure [16].A good feature is depicted by the one where two data points are close to each other if there is an edge between these two points, and thus the feature tends to have small LS. Since features are evaluated independently, selecting *k* features can be achieved by greedily picking the top *k* features which have the minimal LS values. For the *r*-th feature f_r, LS_r is calculated as follows:

$$\mathbf{LS_r} = \frac{\widetilde{\mathbf{f_r}}^{\mathbf{T}}\mathbf{L}\widetilde{\mathbf{f_r}}}{\widetilde{\mathbf{f_r}}^{\mathbf{T}}\mathbf{D}\widetilde{\mathbf{f_r}}} \;(2) \quad \mathbf{where}\;\; \widetilde{\mathbf{f_r}} = \mathbf{f_r} - \frac{\mathbf{f_r^T D1}}{\mathbf{1^T D1}}\mathbf{1}\;\;(3)$$

$$\mathrm{f_r} = [\mathrm{f_{r1}} \quad \mathrm{f_{r2}} \dots \quad \mathrm{f_{rm}}]^{\mathrm{T}}, \mathrm{D} = \mathrm{diag}(\mathrm{S1}), 1 = [1 \quad \dots \quad 1]^{\mathrm{T}}, \mathrm{L} = \mathrm{D} - \mathrm{S}$$

where f_{ri} is the *i*-th sample of the *r*-th feature, *S* is the weight matrix of the graph, and *L* is the graph Laplacian matrix. It is noted that *LS* outperformed data variance and Fisher score methods.

4. RELIEF-F

RELIEF algorithm selects features that contribute to the separation of samples from different classes, by estimating features according to how well their values distinguish among instances that are near each other [19][43]. For that purpose, RELIEF randomly samples *m* instances from the training set and updates the relevance estimation of each feature based on the difference between the selected instance, the nearest instance of the same class (nearest hit), and the nearest instance of the opposite class (nearest miss); estimating the weight of the attribute as follows:

$$\mathbf{W[A]} = \mathbf{W[A]} - \frac{\mathbf{diff(A,R,H)}}{\mathbf{m}} + \frac{\mathbf{diff(A,R,M)}}{\mathbf{m}}\;(4)$$

where *diff (A, R, H/M)* is the *diff (Attribute, Instance R, nearest hit H/nearest miss M)* that is the difference between the values of an attribute for two instances, and *m* is number of instances used for normalization.

RELIEF was extended to search for *k*-nearest hits and misses and to support multiclass through estimating the ability of features to separate each pair of classes regardless of which two classes are closest to each other. Thus, RELIEF-F finds one near miss *M(C)* for each different class and averages their contribution for updating estimates *W[A]*. The average is weighted with the prior probability of each class as follows:

$$\mathbf{W[A]} = \mathbf{W[A]} - \frac{\mathbf{diff(A,R,H)}}{\mathbf{m}} + \sum_{\mathbf{C \neq class(R)}} \frac{\mathbf{[P(C) * diff(A,R,M(C))]}}{\mathbf{m}} \quad (5)$$

RELIEF is able to deal with data sets of dependent and independent attributes, whereas RELEIF-F deals with noisy, incomplete, and multi-class data set.

5. SPEC

SPEC (SPECtrum) presents [46][47] a unified framework for feature selection based on looking at the algorithm not as with or without class labels but as an effort to select features consistent with the target concept related to dividing instances into well separable subsets according to the pairwise instance similarities *S*, where the features are evaluated independently. SPEC employs the spectrum of the graph to measure feature relevance and realizes spectral feature selection according to the structures of the graph induced from *S*. The selection of features is done through building the pairwise instance similarity set *S* to represent the relationships among instances according to the geometric structure of the data or the class affiliation, constructing its graph representation, evaluating features using the spectrum of the graph, and ranking the features in terms of feature relevance that is based on three ranking criteria for measuring feature relevance. The *k*-features are selected based on the ranking that is done in ascending order for criteria 1 and 2 (equation 6 and 7) and descending order for the third criterion (equation 8). The three criteria are as follows:

$$\mathbf{SS_1(F_r)} = \hat{\mathbf{f}}_\mathbf{r}^\mathbf{T} \boldsymbol{\gamma}(\boldsymbol{\mathcal{L}}) \hat{\mathbf{f}}_\mathbf{r} = \sum_{\mathbf{j=0}}^{\mathbf{n-1}} \boldsymbol{\alpha}_\mathbf{j}^\mathbf{2} \boldsymbol{\gamma}(\boldsymbol{\lambda}_\mathbf{j}) \quad (6)$$

$$\mathbf{SS_2(F_r)} = \frac{\hat{\mathbf{f}}_\mathbf{r}^\mathbf{T} \gamma(\mathcal{L}) \hat{\mathbf{f}}_\mathbf{r}}{\mathbf{1} - \hat{\mathbf{f}}_\mathbf{r}^\mathbf{T} \boldsymbol{\xi}_\mathbf{o}} = \frac{\sum_{\mathbf{j=1}}^{\mathbf{n-1}} \alpha_\mathbf{j}^\mathbf{2} \gamma(\lambda_\mathbf{j})}{\sum_{\mathbf{j=1}}^{\mathbf{n-1}} \alpha_\mathbf{j}^\mathbf{2}} \quad (7)$$

$$\mathbf{SS_3(F_r)} = \sum_{\mathbf{j=1}}^{\mathbf{k-1}} (\boldsymbol{\gamma}(\mathbf{2}) - \boldsymbol{\gamma}(\boldsymbol{\lambda}_\mathbf{j})) \boldsymbol{\alpha}_\mathbf{j}^\mathbf{2} \quad (8)$$

where *W* is the adjacency matrix of the graph, *D* is the degree matrix of the graph, *L* = *D*-*W* is the Laplacian matrix, and $\mathcal{L} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$ is the normalized Laplacian matrix. As for the feature notations, f_r is the feature vector of F_r, $\tilde{f_r} = (D^{\frac{1}{2}} f_r)$ is the weighted feature vector of *F*F_r, and $\hat{f_r} = \frac{\tilde{f_r}}{\|\tilde{f_r}\|}$ is the normalized weighted feature vector of F_r. (λ_j ξ_j) are the eigenvalue and eigenvector of $\mathcal{L}$ with ($\lambda_0 = 0$, $\xi_0 = D^{\frac{1}{2}}$e) the trivial eigenpair of the graph having $\mathrm{e} = \{1,1,\dots 1\}^T$ and $\alpha_j = \cos\theta_j$ where θ_j is the angle between f_r and ξ_j. The eigenvectors of $\mathcal{L}$ are related to a Fourier basis and extend the usage of $\mathcal{L}$ to $\gamma(\mathcal{L}) = \sum_{j=0}^{n-1} \gamma(\lambda_j) \xi_j \xi_j^T$ where $\gamma(\lambda_j)$ is an increasing function that penalizes high frequency components, and can be very helpful in a noisy learning environment.

The three feature ranking functions derive families of supervised and unsupervised feature selection in a unified manner, where Relief-F is a special case of the first ranking criterion and LS is a special case of the second ranking criterion. It is noted that SPEC achieves better accuracy than LS and ReliefF.

6. Pearson Correlation

Pearson Correlation is used to assess the correlation between two features according to the coefficient [44] defined as:

$$\mathbf{p} = \frac{\mathbf{1}}{\mathbf{n-1}} \frac{\sum_{i=1}^{n}(\mathbf{x_i} - \mathbf{m_{x_i}})(\mathbf{y_i} - \mathbf{m_{y_i}})}{\boldsymbol{\sigma_{x_i}\sigma_{y_i}}} \quad (9)$$

where x_i and y_i are two variables, with their means m_{x_i} and $m_{y_i,}$ and their standard deviations σ_{x_i} and σ_{y_i} respectively, and *n* the data sample size. If the result is 0, then the two features are independent.

7. Chi-Square

Chi-Square Score is used to discretize the numeric attributes and test for independence to assess whether the class label is independent of a particular feature [20]. The score for a feature with *r* different values and *C* classes is defined as:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{C} \frac{(n_{ij} - \mu_{ij})^2}{\mu_{ij}} \quad (10)$$

where n_{ij} is the number for samples with the *i*-th feature value and:

$$\mu_{ij} = \frac{n_{*j} n_{i*}}{n} \quad (11)$$

where n_{i*} is the number of samples with the *i*-th value for a specific feature, n_{*j} is the number of samples in class *j*, and *n* is the number of samples. Whenever correlation is calculated between a feature and a class, it is known as C-correlation; whenever it is calculated between two features it is known as F-correlation.

8. Gini Index

Gini Index is used for quantifying a feature's ability to distinguish between classes [39]. Gini Index of each feature is calculated independently, and the top *k* features with the smallest Gini index are selected. Given *C* classes, Gini Index of a feature *f* is:

$$\mathbf{GiniIndex(f) = 1 - \sum_{i=1}^{C}[p(i|f)]^2} \quad (12)$$

9. Symmetric Uncertainty

Information Gain is characterized by the reduction in the uncertainty of random variable *X* due to the knowledge of random variable *Y*. In other words, it is the measure of the amount of information that *Y* contains about *X*, or a measure of information flow from *X* to *Y* [33]. It is the amount by which the entropy of *X* decreases reflects additional information about *X* provided by *Y*. The information gain of two random variables *X* and *Y* is defined as:

$$\mathbf{IG(X|Y) = I(X;Y) = H(X) - H(X|Y) = \sum_i \sum_j p(x_i, y_j) \log \frac{p(x_i,y_j)}{p(x_i)p(y_j)}} \quad (13)$$

Here $H(X)$ is the entropy of *X*, the measure of the uncertainty of *X*, and $H(X|Y)$ is the entropy of *X* knowing *Y*.

$$\mathbf{H(X) = -\sum_i P(x_i) \log_2(P(x_i))} \quad (14)$$

$$\mathbf{H(X|Y) = -\sum_j P(y_j) \sum_i P(x_i|\, y_j) \log_2(P(x_i|y_j))} \quad (15)$$

It should be noted that $\mathrm{IG}(X|Y) = \mathrm{I}(X;Y) = \mathrm{I}(Y;X) = \mathrm{H}(Y) - \mathrm{H}(Y|X)$; thus, *Y* says about *X* as much as *X* says about *Y*.

But information gain is biased towards features with more values. Hence the symmetric uncertainty is used instead [37] and defined as:

$$\mathbf{SU}(\mathbf{X},\mathbf{Y}) = \mathbf{2}\left[\frac{\mathbf{IG}(\mathbf{X}|\mathbf{Y})}{\mathbf{H}(\mathbf{X})+\mathbf{H}(\mathbf{Y})}\right] \quad (16)$$

where *IG(X|Y)* is the information gain of *X* knowing *Y* and *H(X)* and *H(Y)* are the respective entropies of *X* and *Y*.

B. Wrapper Model

A wrapper model [41] "wraps" the feature selection process around a learning algorithm. Instead of ranking features independently, it uses the feedback from a learning algorithm to determine which features to keep. A feature selection algorithm is categorized as a wrapper model if it is directly related to the performance of a learning model, usually in terms of its predictive accuracy. Moreover, wrapper models aim at selecting subsets of variables that together have a good predictive power, as opposed to ranking variables according to individual predictive power. Common examples for feature subset search include Sequential Forward Stepwise selection (SFS) [18], Sequential Backward Stepwise selection (SBS) [18], Plus-l-Minus-r method (LRS) [45], Sequential Floating Search Method (SFSM) [38], Branch and Bound method [35], and Genetic Algorithm (GA) [4].

1. Sequential Forward Stepwise Selection (SFS)

SFS [18] starts with no features and add them one by one, at each step adding the one that decreases the error the most or that maximizes an objective function in combination with the previous feature set, until any further addition does not significantly decrease the error or maximizes the objective function.

2. Sequential Backward Stepwise Selection (SBS)

SBS [18] tries to exclude one redundant feature at a time from the current feature set. It starts with all the features and removes them one by one, at each step removing the one that decreases the error the most or that induces the smallest decrease of the objective function.

3. Plus-l-Minus-r Method (LRS)

Plus-l-Minus-r [45] is a generalization of SFS and SBS that avoids nesting of feature sets by involving successive augmentation and depletion processes. If L>R, LRS starts from the empty set and repeatedly adds ‘L’ features and removes ‘R’ features. If L<R, LRS starts from the full set and repeatedly removes ‘R’ features followed by ‘L’ feature additions.

4. Sequential Floating Search Method (SFSM)

SFSM [38] is the evolution of SBS and SFS and the extension to the LRS algorithm where flexible values for L and R are determined automatically from the data and updated dynamically at each step. Having the values of L and R "floating" in order to approximate the optimal solution as much as possible, the nesting effect that SBS and LRS suffer from is avoided. The nesting effect states that some features with bad performance could perform well while combining with other features. Two versions of SFSM exist: The SFFS (sequential forward floating selection) applies SFS starting from the current feature set, followed by a series of successive conditional exclusion of the worst feature in the newly updated set provided a further improvement can be made to the previous sets. The SBFS (sequential backward floating selection) applies SBS starting from the current feature set, followed by a series of successive conditional inclusions of the most significant feature from the available features if an improvement can be made to the previous sets. Overall, the floating methods perform better than their non-floating counterparts, giving near-optimal results with reasonable execution times.

5. *Branch and Bound*

The branch and bound algorithm [35] is considered to be an optimal feature selection method that is known for its computational efficiency. It avoids exhaustively exploring the entire search space by rejecting many subsets that are guaranteed to be suboptimal without direct evaluation through the criterion function that satisfies the monotonicity condition. The monotonicity condition states that whenever the criterion evaluated for any node is less than a bound, all successors of that node also have criterion values less than the bound and cannot be the optimum solution. Branch and bound starts from the full set and removes features using a depth-first strategy when fining the nodes whose objective function are lower than the current best.

$$\mathbf{P(C_i|A_1, \ldots, A_n) = P(C_i)P(A_1|C_i) \ldots P(A_n|C_i)/P(A)} \quad (17)$$

6. *Genetic Algorithm*

Genetic algorithm is a stochastic local search algorithm inspired by the Darwinian evolution theory, which is well known to address complex problems in Search-Based

Software Engineering [26][36]. GA solves a given problem by operating on an initial population of candidate solutions or chromosomes, evaluating their quality using a fitness function, applying a form of transformation (selection, crossover, mutation, replacement) to form new generations and improve the quality of these solutions, and ultimately evolving to a single solution or set of solutions that fit certain criteria.

The initial population can be chosen randomly or it can be specific to the problem in hand.

- The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions.
- The selection process is usually based on the roulette wheel procedure to select the fit parent for the next generation since selecting the more fit parents for generation will insure that the next generation will have a higher average fitness and thus serving as a better candidate.
- The crossover procedure is applied to create the offspring. The well-known crossover technique is the single-point crossover where the two parent chromosomes are broken at a random position to create the two offspring by exchanging the broken portions. For example the first child chromosome is the concatenation of the first part of the first parent with the second part of the second parent. The aim is to create a new chromosome that carries a mixture of the parents' characteristics which might lead to better solutions, besides seeking to increase diversity among population by mixing information from parent individuals.

- Mutation is applied on the new chromosome by deforming it through flipping each of its bits with a certain probability, seeking to maintain genetic diversity and avoiding being stuck in a local optimum where most of the solutions are similar.
- For the replacement stage, the steady-state replacement approach can be adopted where the population of solution is modified by replacing the previous solution with the newly created one where each offspring resulting from a crossover operation replaces a less fit individual of the population.
- The stopping criteria can be determined by the number of generations, iterations executed, or when a fitness criterion is met.

C. Hybrid Model

A hybrid model is meant to combine a filter model and a wrapper model. In other words, it combines the strength of filters and wrappers while avoiding their drawbacks. Based on previous work, it typically involves applying a filter model on the original dataset, then the result is further processed using a wrapper model [4][17]. The filter model is usually applied first to define a small space of potentially informative features. The wrapper model comes next to refine the selection process by optimizing the accuracy.

In [4], a filter approach was used to define multiple feature spaces, where each feature space was explored through a GA that acted as a wrapper selector. Then, the information from the different spaces was fused to identify the most relevant features.

In [17] for example, two filter models were used to screen out redundant features, and the resulted feature subsets were combined for the wrapper method to do final fine tuning.

CHAPTER IV

PROPOSED REDUCTION TECHNIQUES

In our work we proposed a filter and a wrapper technique to assess which is able to perform a safe reduction of execution profiles. Furthermore, a hybrid model was created, which is the combination of the filter and wrapper model to study its effect. Concerning the filter model, the symmetric uncertainty measure was adopted for the proposed reduction technique. As for the wrapper model, the genetic algorithm was used. The following subsections explain in detail the proposed reduction techniques.

A. Filter Model

In our proposed filter-based reduction technique, we opted to investigate the information-theoretical approach since it is widely used in the data mining literature. We decided to use the *SU* measure since it avoids the bias that information gain incurs. Furthermore, this measure is symmetric in nature, where *SU(X, Y)* is same as that of *SU(Y, X),* which halves the number of needed computations during analysis. In addition, this measure gives normalized values in the range [0, 1], ensuring they are comparable, with the value 1 indicating that knowledge of the value of either *X* or *Y* completely predicts the value of the other and the value 0 indicating that *X* and *Y* are independent. For these reasons, we decided to use the symmetric uncertainty measure. We actually explored two variants of the *SU* algorithm, seeking to choose the version that would give the best assessment, as described next.

1. SU_1

In the *SU_1* approach, the symmetric uncertainty measure is used to assess the relevance of the feature to the class labels along with assessing the redundancy between features. For the purpose of software analysis, it is not always the case that the class labels are available; actually, in software testing for example, the goal is to determine these class labels (i.e., to classify test cases as *passing* or *failing*). *SU_1* overcomes the lack of class labels by approximating them as follows:

Step1: Using the initial execution matrix of the profiling elements, *k-means* clustering is performed while varying *k* (number of clusters) from 3 to 10. This range is chosen due to the following assumptions about the subject programs in our study: they contain multiple faults but not overly a large number of faults since they are small in size and are stable releases.

Step2: In order to assess the quality of the obtained clusters, we use the *Davies-Bouldin index* (DBIndex) as a measure [14], which does not depend on the class label information, but is based on a ratio of intra-cluster and inter-cluster distances, where a lower *DBIndex* implies a better clustering.

- *DBIndex:*

It is an internal cluster evaluation measure, based on the data that was clustered itself; the validation of the clustering is based on the quantities and features inherent to the dataset [14]. *DBindex* is based on a ratio of within-cluster and between-cluster distances.

$$\mathbf{DBIndex} = \frac{1}{k}\sum_{i=1}^{k} \max_{i \neq j} \left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}\right) \quad (18)$$

where k is number of clusters, c_i and c_j the centroids of clusters i and j respectively, σ_i is the average distance of all elements in cluster i to centroid c_i, σ_j is the average distance of all elements in cluster j to centroid c_j , and $d(c_i , c_j)$ is the distance between centroids.

From the definition, we can recognize that a lower *DBIndex* implies a better clustering. A main drawback of this method is that a good, low value does not imply the best information retrieval, and hence we will not use this index for evaluating our reduction techniques. However, this index serves as a good measure of the number of clusters the data could be ideally classified into, and hence we will use it to get the *Class Labels Vector* (CLV).

Step3: The number of clusters k_{best} that exhibits the best (lowest) DBIndex is identified, i.e., k_{best} yields the best quality clusters in terms of low intra-cluster distances and high inter-cluster distances. The k_{best} clusters are then used to form the k_{best} class labels, which is an attempt to associate each test case with its true class label, e.g., *passing*, *failing due to fault-1*, *failing due to fault-2*, etc.

The resulting class labels are used by SU_1 to reduce the execution profiles according to the pseudo-code below. Hereafter, a *Class Labels Vector* (CLV) captures the class labels information. Given k classes and an execution matrix with N execution profiles (N test cases), a CLV contains a sequence of N integers in the range [1, k] each referring to a class label.

```
Input: Execution Matrix M, CLV C, Threshold T
Output: Reduced Matrix M'

1. //SU between feature and class
2. For each feature $F_i \subset M$
3.   Calculate the symmetric uncertainty $SU(F_i; C)$
4.   between the respective feature Fi and the CLV C.
```

```
5.   If SU($F_i$; C) != 0
6.    Add Fi to M’
7.   end
8. end
9. //SU between feature and feature
10. For each feature $F_i \subset$ M’
11.  For each feature $F_j \subset$ M’ where j > i
12.   If $F_i$ is not removed from M’ && $F_j$ is not removed from M’
13.    Calculate the symmetric uncertainty SU($F_i$; $F_j$)
14.    between the feature $F_i$ and $F_j$.
15.    If SU($F_i$; $F_j$) >= T
16.     If SU($F_i$,C) > SU($F_j$,C)
17.      Remove $F_j$ from M’
18.     else
19.      Remove $F_i$ from M’
20.     end
21.    end
22.   end
23.  end// loop at line 11
24. end// loop at line 10
25. return M’
```

Lines 2-8: For every feature in the input matrix *M*, the symmetric uncertainty is calculated between the feature and *C* to measure its relevance. A feature that yields a value of zero is considered *completely irrelevant*.

Lines 5-7: Features that are irrelevant to *C* are ignored, and the rest are added to the reduced matrix *M’*.

Lines 10-25: Symmetric uncertainty between features is calculated to measure their degree of redundancy. (Note that two features that yield a value of 1 are considered *completely redundant*).

Lines 10-11: This line restricts the calculation of the symmetric uncertainty between features to those in the reduced matrix *M’*. Only features with j > i are considered due to the symmetric nature of the SU, thus incurring less computational cost.

Line 12: A check is made to ensure that the two features to be compared for redundancy are still present in *M'*, due to the possibility of anyone getting removed at a later stage in the algorithm.

Line 15: The computed symmetric uncertainty between features is compared relative to a predefined threshold *T*. If the value is greater than *T*, the two features are considered redundant to some extent where one feature can represent the other.

Line 16: To decide which feature to keep and which to ignore, the feature with more relevance to *C* is kept.

Lines 17-19: Hence the feature with a greater SU with *C* is kept and the other is removed from *M'*. The action of removing a feature from *M'* explains why the check in *Line 12* is needed.

Line 25: The reduced matrix *M'* is returned.

Given test suite *T* and set of features *E*, the cost of $SU(F_i; F_j)$ is $O(T^2)$ since the cost of $IG(X|Y)$ is $O(T^2)$ and that of $H(X)$ is $O(T)$. In the worst case, all features are considered and none has been removed, yielding a runtime cost for SU_1 of $O(|T|^2.|E|^2)$.

Finally, in the experiments presented in the next two chapters, the value of the threshold *T* was varied and it was observed that compared to 0.8, 0.9, 0.95 and 0.97, a $T = 0.95$ performed best.

2. *SU2*

Having concerns that the approximated classification adopted in SU_1 might not be accurate; in SU_2, we omitted the use of class labels altogether. This approach cannot check for irrelevant features due to the absence of class labels, but will assess the pair-wise redundancy of a given feature with all the other features, and will also assess how redundant this feature is to the set of features as a whole. The pseudo-code below shows how SU_2 reduces execution profiles.

```
Input: Execution Matrix M, Threshold T
Output: Reduced Matrix M

1. //SU between feature and feature
2. For each feature $F_i \subset M$
3.  sum=0, count=0;
4.  For each feature $F_j \subset M$ where j != i
5.   If $F_i$ is not removed && $F_j$ is not removed from M
6.    Calculate the symmetric uncertainty SU($F_i$;$F_j$)
7.    between the feature $F_i$ and $F_j$.
8.    If SU($F_i$;$F_j$) == 1
9.     Remove $F_i$ from M;
10.    break;
11.   else
12.    count++;
13.    sum += SU($F_i$;$F_j$);
14.   end
15.  end
16. end // loop at line 4
17.
18. If $F_i$ is not removed from M
19.  average = sum/count;
20.  If average > T
21.   remove Fi from M
22.  end
23. end
24.end// loop at line 2
25.return M
```

Line 2: Every feature in the input matrix *M* is considered.

Line 4: Given F_i, every other feature F_j is considered where j != i. Note that in SU_1 only features with j > i are considered, this dissimilar decision is necessitated by *Lines 11-14*.

Line 5-10: For a given feature F_i, the symmetric uncertainty is calculated with every other feature. *Fi* is removed from *M* if the score = 1, i.e., the feature is *completely redundant*. A score of 1 occurs when a feature F_i is the exact copy of feature F_j or is the exact opposite of feature F_j, e.g., <0, 1, 1> is the opposite of <1, 0, 0>.

Line 11-14: In case of no complete redundancy, *sum* and *count* are updated. The summation of all symmetric uncertainties between F_i and every other feature still present in *M* is stored in *sum*. The number of features currently accounted for is stored in *count*.

Line 18-23: If F_i is still in *M*, the *average* symmetric uncertainty for F_i with all other features is computed and compared to a threshold *T*. If the average is greater than *T*, then F_i is removed from *M* as it is considered to be redundant to the entire feature set.

Line 25: The reduced matrix is returned.

It should be noted that the *T* used in SU_1 represents the threshold for the symmetric uncertainty between two features; whereas in SU_2 it represents the threshold for the average symmetric uncertainty for a given feature with all other features.

Here also, in the experiments, the value of *T* was varied and it was observed that compared to 0.1, 0.2, and 0.3, a *T* = 0.1 performed best. Finally, similar to SU_1, SU_2 has a runtime cost of $O(|T|^2.|E|^2)$.

B. Wrapper Model

In our proposed wrapper-based reduction technique, we opted to adopt the genetic algorithm for two reasons: a) it is widely used in the data mining literature; and b) it allows for a methodical way to control the selection process via the use of a fitness function. In our work, the fitness function will measure the prediction performance of our genetic learning algorithm, while aiming for a fitness of 1. Next we present the GA based reduction technique described in [2], then present an enhanced version of it that uses a different fitness function that leverages program heuristics.

1. GA₁

Assuming the set of program elements being considered is *E*, the problem of determining the most representative subset in terms of execution status translates to the task of finding a subset *S* that results in $H(\bar{S}/S)$ = 0, where $\bar{S} = E - S$ and $H(\bar{S}/S)$ is the entropy of $\bar{S}$ given *S*, i.e., if the execution state of *S* is known then the execution state of its complement $\bar{S}$ will be known. This is equivalent to finding a subset *S* such that |*DV(S)*| = |*DV(E)*|, where *DV(S)* is the set of distinct values assumed by *S*, and *DV(E)* the set of distinct values assumed by *E*.

Of course, *E* is one such subset but we are interested in those whose size is minimal. Given the size of the search space at hand ($2^{|E|}$), we opt to use a heuristic approach to search for potential representative subsets. Specifically, we use a genetic algorithm where each candidate solution is represented by a vector of bits (*chromosome*) whose size is equal to the total number of elements. A value of 1 means that the corresponding element is included in the solution and a value of 0 indicates otherwise. The fitness of a particular solution/subset S is quantified as $fitness(S) = \frac{|DV(S)|}{|DV(E)|}$. The pseudocode below describes our technique which takes an execution matrix M associated with a set of program elements E as input and determines a (likely small) subset of E with fitness equal to 1. As a pre-processing step, we perform basic redundancy removal to arrive at a reduced matrix *M'* and element set *E'*, which is useful to reduce the search space. The genetic algorithm first creates an initial population by randomly generating small subsets of *E'*. After that, it repeatedly applies crossover and mutation to produce new solutions. Every time a superior solution emerges, it replaces a less fit one in the population. This iterative process is terminated in two cases: 1) a solution having a fitness of 1 is encountered or 2) the maximum number of iterations is exhausted. In case the GA terminates without arriving at a fitness of 1, we augment the best encountered solution by adding one element at a time in a greedy fashion so as to reach the maximum fitness. One might argue that such step could be done starting from an empty solution or a totally random one. However, such approach has two disadvantages. First, it's very costly if the starting solution has low fitness. Second, as the size factor isn't enforced, this approach wouldn't likely yield small solutions. Next we detail the steps of our algorithm.

```
Input: Execution Matrix M, Profiling Elements E
Parameters: MAX_ITERATIONS //num of GA iterations
        POP_SIZE //population size
        MUT_PROB //probability of mutation
Output: A chromosome representing a subset of E with fitness equal to 1

1. (M', E') ← BasicRedundancyRemoval(M, E)
2. nbIterations ← 0
3. result ← null
4. population ← genRandomSubsets(E',POP_SIZE)
```

```
5. while [nbIterations < MAX_ITERATIONS] AND
        [fit(result) < 1.0]
6.   (p1, p2) ← rouletteWheelSelection(population)
7.   child ← crossover(p1, p2)
8.   child ← mutation(child, MUT_PROB)
9.   replace(population, p1, p2, child)
10.  best ← getSolutionWithMaxFitness(population)
11.  if fit(best) > fit(result)
12.      result ← best
13.  end if
14.  nbIterations ← nbIterations + 1
15. end while
16. while [fit(result) < 1]
17.      result ← result ∪ maxGain(result, E'-result)
18. end while
19. return result
```

Line 1: Basic redundancy removal is applied on M and E. That is, E is partitioned into E1, E2, …, En where each Ei contains elements of E having equivalent columns in M. E' is formed by choosing one element from each Ei and M' is derived from M by removing the columns corresponding to the elements in E-E'.

Lines 2-3: Some variables are initialized; e.g., result is used to keep track of the best encountered solution.

Line 4: The initial population is built by generating random subsets of E' whose sizes are close to$\log_2|\mathrm{DV(E)}|$. Such choice is guided by the fact that the size of the smallest possible representative subset having a fitness of 1 cannot be smaller than this threshold. This step is an important factor to converge subsequently to a relatively small solution. The size of the initial population, which is maintained in subsequent iterations, is equal to POP_SIZE.

Line 5: The algorithm loops MAX_ITERATIONS times unless a solution with fitness 1.0 is encountered.

Line 6: Within each iteration, the algorithm selects two parent chromosomes using the roulette wheel methodology, which randomly selects one chromosome at a time based on its relative fitness with respect to the population. i.e., solutions with higher fitness values are more likely to be selected.

Line 7: The selected parent chromosomes undergo single-point crossover to create a child chromosome as follows. First, each of them is split at the same random position. Then, a new (child) chromosome is created by concatenating the first part of the first parent with the second part of the second parent.

Line 8: The child chromosome undergoes mutation, where each bit is randomly flipped with a probability equal to MUT_PROB.

Line 9: The child chromosome replaces the less fit parent if the fitness of the former is higher than that of the latter.

Lines 10-12: The result is constantly updated by comparing it to the best solution obtained in every subsequent generation.

Lines 16-18: If the solution returned by the GA (say SGA) doesn't have a fitness of 1.0., we augment it by adding one element at a time from E'-SGA until the fitness becomes 1.0. Each time we add the element that results in the maximum increase in fitness.

As for the parameters of the GA, we used a value of 1000 for MAX_ITERATIONS, 0.05 for MUT_PROB, and 100 for POP_SIZE. Finally, the runtime cost of GA_1 is $O(|T|.|E|^2)$ where T is the test suite, and E is the set of features. In the worst case, the columns in the original execution matrix would be pair-wise distinct and the augmentation step (lines 16-18) would include all elements.

2. *GA2*

GA_2 differs from *GA_1* by having each candidate solution being assessed based on a specific program heuristic at different stages of the algorithm. We will first present this heuristic and then explain how it is incorporated into our algorithm. The heuristic conjectures that a program element that is not covered by many tests should be considered more relevant to failure and assigned a higher score, since in typical test suites the number of failing test cases is much smaller than the number of passing test cases [9]. Given a test suite *T* and a program element *pe_i*, the score assigned to *pe_i* is *$Score(pe_i)$*$= 1 - \frac{count(pe_i)}{|T|}$, where *count*(*$pe_i$*) is the number of tests covering *pe_i*; and if *pe_i* never occurred in any of the test cases then *Score* (*pe_i*) is 0. In *GA_2*, the relevance to failure of a potential solution (*chromosome*) is computed according to the following equation:

$$\mathbf{weight(solution)} = \frac{\sum_{\mathbf{i=1}}^{\mathbf{E}} \mathbf{10}^{\mathbf{Score(pe_i)}}}{\mathbf{count(pe_i)*10}} \quad (19)$$

In the above, for a program element *pe_i* to be considered in the computation, it should have a bit value of 1 in the candidate solution, and it should have occurred in at least one test case. Note how to emphasize larger scores, we used an exponential weighing approach (where 10 was arbitrary chosen). The weight of a solution is used in the following cases:

- When creating the initial population, only solutions with weights higher than a given threshold are kept.

- When applying crossover and mutation to produce a child solution, the weight of the new solution is compared to a threshold to decide whether to keep it or not. The threshold used in this and the previous case was chosen to be 0.3 based on experimental trials.
- When comparing the fitness of two solutions S_1 and S_2, the weights of the solutions are used to break the tie in case *fitness*(S_1) and *fitness*(S_2) were equal. The solution with the higher weight is kept, i.e., the one that is more relevant to failure.

Note that the runtime cost of GA_2 is $O(|T|.|E|^2)$ similar to that of GA_1.

C. Hybrid Model

As described earlier a hybrid model is meant to combine a filter model and a wrapper model. Typically, it involves applying a filter model on the original dataset, and then the result is further processed using a wrapper model [4][17]. In this work, we adopted a different approach. Since the two models identify significant features following different and potentially complementary strategies, in our approach we let each model process the original dataset independently and then join the resulting reduced datasets by either computing their intersection or their union. During our experiments involving SU_1, SU_2, GA_1, and GA_2 (sections 4 and 5), we identified the better performing filter model and wrapper model, which turned out to be SU_1 and GA_2. Based on that finding we devised the following two hybrid models:

- $H_{SU1 \cap GA2}$: involves running SU_1 and GA_2 then computing the *intersection* of their respective resulting features. The aim here is to retain the features that have been classified as indispensable by both models.
- $H_{SU1 \cup GA2}$: involves running SU_1 and GA_2 then computing the *union* of their respective resulting features. Since each model operates following a different strategy, here we assume that the respective strategies of SU_1 and GA_2 are complementary. Note that after unionizing both sets, the *completely redundant* features are removed.

CHAPTER V

QUALITY OF REDUCTION ASSESSMENT

The assessment of the reduction techniques was performed in terms of the reduction rate, the information loss measure, the impact on the quality of cluster analysis in general, and the runtime cost. First, we describe our subject programs before we present the different assessments.

A. Subject Programs and Test Suites

The subject programs that we used in our experiments are: 1) *tot_info*, *schedule*, *print_tokens2*, *space*, *flex2*, and *sed3* from the SIR repository [48] and 2) *Tomcat 3.0*, *Tomcat 3.2.1*, *Jigsaw 2.0.5*, and *JTidy* that we used in previous work [26][31][34]. Except for *space*, *flex2*, and *sed3*, which are written in C, all others are Java programs; note that *tot_info*, *schedule*, *print_tokens2* are Java programs that we converted from C in previous work [27]. The features that we considered in this work are program elements of one of three types, namely, *BB, ALL,* or *SliceP*. Where *ALL* combines *BB, BBE,* and *DUP* program elements, which are all described below along with *SliceP*:

- *BB* (basic blocks): For every basic block *B* such that *B* is executed in at least one test case, a *BB* feature equals to 1 indicates that *B* is executed in the current test.
- *BBE* (basic-block edges or branches): For every pair of basic blocks *B1* and *B2* such that there is a branch from *B1* to *B2* in at least one test case, a *BBE* feature equals to 1 indicates that this branch is taken in the current test.

- *DUP* (def-use pairs): For every pair consisting of a variable definition *D*(*x*) and a use *U*(*x*) such that *D*(*x*) dynamically reaches *U*(*x*) in at least one test case, a *DUP* feature equals to 1 indicates that *D*(*x*) dynamically reaches *U*(*x*) in the current test.
- *SliceP* (slice pairs): For each pair of statements s_1 and s_2 such that s_1 occurs in a backward dynamic slice on s_2 in at least one test, a *SliceP* profile contains a count of how many times s_1 occurs in such a slice.

Table 5 lists the subject programs along with their respective test suite sizes, number of faults, and number of features. Note that the subscript at the end of each program name indicates the type of feature used in that particular instance. In seven instances we used execution profiles of type *ALL*, in three instances we used *BB*, and in only one instance, involving *JTidy*, we used *SliceP* profiles. This choice was made based on the following:

- *ALL* profiles were used in most cases since execution profiles typically comprise program elements of type *BB*, *BBE*, and *DUP*.
- *BB* profiles were used for *space*, *sed3*, and *flex2* due to the lack of availability of profilers supporting C programs (our in house profilers only support Java programs).
- *SliceP* profiles were used only once since these profiles are very costly to collect and process, e.g., given our computing resources, we were not able to collect and process more than 475 *SliceP* profiles for *JTidy*. Also note that *SliceP* profiles are expected to be large in size due to the large number of indirect data and control

dependences in programs, which is clearly noticeable when comparing the numbers of features associated with $JTidy_{ALL}$ and $JTidy_{SliceP}$ (22,110 vs. 1,431,487).

In order to generate the execution profiles for the Java programs, the programs were instrumented and profiled using profilers that we developed in previous work [34]. Whereas the three C programs were profiled using GCov to collect statement profiles only (we assume that the information contained in statement profiles and *BB* profiles are equivalent for our purposes).

Table 5. Subject Programs

Programs	# of Faults	# of Test Cases	# of Features
$print\_tokens2_{ALL}$	7	2349	891
tot_info_{ALL}	6	939	1,276
$Schedule_{ALL}$	3	2295	1,047
$Space_{BB}$ *(C)*	5	2000	3,164
$Tomcat3.0_{ALL}$	4	658	26,137
$Tomcat3.2.1_{ALL}$	3	497	24,438
$Jigsaw_{ALL}$	4	530	29,895
$JTidy_{ALL}$	3	1000	22,110
$flex2_{BB}$ *(C)*	3	531	2,914
$sed3_{BB}$ *(C)*	3	195	1,328
$JTidy_{SliceP}$	3	475	1,431,487

Finally, in order to measure the impact of reduction more accurately, the test suites were modified as follows:

- Failing test cases caused by more than one bug were discarded to eliminate the confusion of what bug triggered the failure of the test case.
- Test cases that exercised the bug but did not fail were discarded; i.e., coincidental correctness was mitigated [25]. This was done to eliminate external factors that might affect our results.

Appendix A shows the detail of the modification each subject program had undergone.

B. Assessment 1: Reduction Rates

Given that our execution profiles are structural in nature (as it is typically the case), they are expected to contain a lot of redundancy mainly due to the transitivity relationships induced by control and data dependences [22].Table 6 summarizes the reduction rates (% of features removed) achieved using our techniques. Clearly, the observed rates are very high, ranging on average from 92.66% to 98.09%. The rate of reduction of $H_{SU1 \cap GA2}$ is highest, and the rates for GA_1 and GA_2 were measurably higher than those of SU_1 and SU_2. Finally, note that the average rate of reduction is 92.73% for the *BB* instances, 96.08% for the *ALL* instances, and 99.67% for *SliceP* instances. One explanation for this observation is that *SliceP* profiles capture program dependences more effectively than *ALL* profiles, and thus are more apt for redundancy removal. The same comparison applies to *ALL* vs. *BB* profiles.

Table 6. Reduction Rates

Program	SU_1	SU_2	GA_1	GA_2	$H_{SU1 \cap GA2}$	$H_{SU1 \cup GA2}$
$print\_tokens2_{ALL}$	89.56%	89.23%	94.50%	94.48%	94.79%	89.25%
tot_info_{ALL}	92.95%	94.12%	96.76%	96.85%	97.21%	92.62%
$Schedule_{ALL}$	84.24%	90.16%	93.68%	93.60%	93.94%	83.90%
$Space_{BB}$	84.17%	83.25%	97.54%	97.58%	97.72%	84.06%
$Tomcat3.0_{ALL}$	99.09%	99.22%	99.80%	99.80%	99.83%	99.06%
$Tomcat3.2.1_{ALL}$	99.33%	99.49%	99.83%	99.84%	99.84%	99.33%
$Jigsaw_{ALL}$	99.26%	99.21%	99.82%	99.83%	99.85%	99.24%
$JTidy_{ALL}$	93.89%	95.52%	99.46%	99.47%	99.53%	93.83%
$flex2_{BB}$	93.10%	92.11%	98.93%	98.99%	99.13%	92.99%
$sed3_{BB}$	85.77%	86.52%	97.29%	97.12%	97.20%	85.75%

$JTidy_{SliceP}$	99.27%	99.50%	99.99%	99.99%	99.99%	99.27%
Average	92.78%	93.49%	97.96%	97.96%	98.09%	92.66%

C. Assessment 2: Information Loss

Assessing a reduction technique based on its reduction rate would be misleading if information loss is not also taken into consideration. We investigated several existing measures for information loss [21]. However, we opted to devise a new metric, which is computationally costly but relatively much more accurate. Our metric, which we call *totalLoss,* is derived below in term of the following entities:

- $comb_{removed}$: a combination of values taken by the removed set of features within the same profiled execution.
- $comb_{reduced}$: a combination of values taken by the reduced/remaining set of features within the same profiled execution.
- *numReducedComb*: number of distinct $comb_{reduced}$ in the reduced set of features.
- $countDistinct(comb_{reduced})$: number of distinct $comb_{removed}$ that $comb_{reduced}$ represents.
- $countAll(comb_{reduced})$: cumulative number of occurrence of all $comb_{removed}$ that $comb_{reduced}$ represents.
- $max(comb_{reduced})$: number of occurrence of the most frequently occurring $comb_{removed}$ that $comb_{reduced}$ represents.
- $loss(comb_{reduced})$: loss associated with $comb_{reduced}$.

$$loss(comb_{reduced}) = \frac{\mathbf{1}}{\boldsymbol{countDistinct(comb_{reduced})}} * \frac{\boldsymbol{max(comb_{reduced})}}{\boldsymbol{countAll(comb_{reduced})}} \quad (20)$$

$$\boldsymbol{totalLoss} = \frac{\boldsymbol{numReducedComb} - \sum_{i=1}^{numReducedComb} \boldsymbol{loss}(\boldsymbol{comb_{reduced}})}{\boldsymbol{numReducedComb}} \quad (21)$$

A *totalLoss* of 0 implies that the reduction technique is lossless and the larger the *totalLoss* the more lossy the technique is.

Next we provide scenarios to better understand the new *totalLoss* metric. In all scenarios we assume that there are 32 execution profiles, the original set of features contains five features $\{f_1, f_2, f_3, f_4, f_5\}$, and the reduced set comprises f_1 and f_3 only.

Scenario 1 - As shown in Table 7, in this scenario: a) in all 9 test cases when $\{f_1, f_3\}$ take on the values {1, 0}, $\{f_2, f_4, f_5\}$ take on {1, 1, 1}; b) in all 12 test cases when $\{f_1, f_3\}$ take on {0, 1}, $\{f_2, f_4, f_5\}$ take on {0, 1, 0}; and c) in all 11 test cases when $\{f_1, f_3\}$ take on {1, 1}, $\{f_2, f_4, f_5\}$ take on {0, 0, 0}.

Table 7. Scenario 1

$comb_{reduced}$ $\{f_1, f_3\}$	$comb_{removed}$ $\{f_2, f_4, f_5\}$
c1: {1, 0}	{1, 1, 1} ×(9)
c2: {0, 1}	{0, 1, 0} ×(12)
c3: {1, 1}	{0, 0, 0} ×(11)

Walking through the steps to compute the value of *totalLoss:*

numReducedComb = 3, *countDistinct*(*c1*) = 1, *countDistinct*(*c2*) = 1, *countDistinct*(*c3*) = 1, *countAll*(*c1*) = 9, *countAll*(*c2*) = 12, *countAll*(*c3*) = 11, *max*(*c1*) = 9, *max*(*c2*) = 12, *max*(*c3*) = 11, *loss*(*c1*) = 1/1 * 9/9 = 1, *loss*(*c2*) = 1/1 * 12/12 = 1, *loss*(*c3*) = 1/1 * 11/11 = 1.

Thus leading to *totalLoss* = (3 - 3)/3 = 0, suggesting a lossless reduction. In reality, this reduction is truly lossless since in all 32 cases, by knowing the values of {f_1, f_3}, the values of {f_2, f_4, f_5} will be known; in other words, every distinct $comb_{reduced}$ maps to a single distinct $comb_{removed}$.

Scenario 2 - The first two rows of Table 8 show the same information as in *Scenario1*, but the third row shows that when {f_1, f_3} take on the values {1, 1}, {f_2, f_4, f_5} take on {0, 0, 0} one time, {1, 0, 1} 5 times, {1, 1, 0} two times, and {0, 1, 1} three times.

Table 8. Scenario 2

$comb_{reduced}$ {f_1, f_3}	$comb_{removed}$ {f_2, f_4, f_5}			
c1: {1, 0}	{1, 1, 1} ×(9)			
c2: {0, 1}	{0, 1, 0} ×(12)			
c3: {1, 1}	{0, 0, 0} ×(1)	{1, 0, 1} ×(5)	{1, 1, 0} ×(2)	{0, 1, 1} ×(3)

The steps to compute the value of *totalLoss* are:

numReducedComb = 3, *countDistinct*(*c1*) = 1, *countDistinct*(*c2*) = 1, *countDistinct*(*c3*) = 4, *countAll*(*c1*) = 9, *countAll*(*c2*) = 12, *countAll*(*c3*) = 11, *max*(*c1*) = 9, *max*(*c2*) = 12, *max*(*c3*) = 5, *loss*(*c1*) = 1/1 * 9/9 = 1, *loss*(*c2*) = 1/1 * 12/12 = 1, *loss*(*c3*) = 1/4 * 5/11 = 0.114.

This yields a *totalLoss* = (3 – 2.114)/3 = 0.296, suggesting a somewhat lossy reduction. This reduction is actually lossy since when the values of {f_1, f_3} are known to be {1, 1}, the values of {f_2, f_4, f_5} cannot be perfectly predicted, since $comb_{reduced}$ *c3* maps to more than one distinct $comb_{removed}$.

Scenario 3 – Following similar steps as in *Scenario 1* and *Scenario 2*. This scenario yields: *loss*(*c1*) = 1/1 * 9/9 = 1, *loss*(*c2*) = 1/1 * 12/12 = 1, *loss*(*c3*) = 1/4 * 8/11 = 0.18, and *totalLoss* = (3 – 2.18)/3 = 0.273; suggesting a slightly less lossy reduction than in *Scenario 2*. This reduction is in fact less lossy since when the values of $\{f_1, f_3\}$ are known to be {1, 1}, the values of $\{f_2, f_4, f_5\}$ can be predicted little more accurately; because in this scenario, the likelihood that $\{f_2, f_4, f_5\}$ will take on the values {1, 0, 1} is 8/11 as opposed to 5/11 in *Scenario 2*.

Table 9. Scenario 3

$comb_{reduced}$ $\{f_1, f_3\}$	$comb_{removed}$ $\{f_2, f_4, f_5\}$			
c1: {1, 0}	{1, 1, 1} ×(9)			
c2: {0, 1}	{0, 1, 0} ×(12)			
c3: {1, 1}	{0, 0, 0} ×(1)	{1, 0, 1} ×(8)	{1, 1, 0} ×(1)	{0, 1, 1} ×(1)

Scenario 4 – Following similar steps as in the previous scenarios. This scenario yields: *loss*(*c1*) = 1/1 * 9/9 = 1, *loss*(*c2*) = 1/1 * 12/12 = 1, *loss*(*c3*) = 1/2 * 10/11 = 0.454, and *totalLoss* = (3 – 2.454)/3 = 0.181; suggesting a less lossy reduction than in *Scenario 3*. Clearly, when $\{f_1, f_3\}$ are known to be {1, 1}, predicting $\{f_2, f_4, f_5\}$ would be more accurate than in *Scenario 3*; since the $comb_{reduced}$ *c3* maps to two distinct $comb_{removed}$ as opposed to four.

Table 10. Scenario 4

$comb_{reduced}$ $\{f_1, f_3\}$	$comb_{removed}$ $\{f_2, f_4, f_5\}$
c1: {1, 0}	{1, 1, 1} ×(9)
c2: {0, 1}	{0, 1, 0} ×(12)

c3: {1, 1}	{0, 0, 0} ×(1)	{1, 1, 0} ×(10)

The above scenarios demonstrate that using *totalLoss* as a measure for information loss seems sensible. Table **11** shows the *totalLoss* values for all the techniques using all subject programs. It can be observed that GA_1, GA_2 and $H_{SU1 \cup GA2}$ are lossless, and the rest are slightly lossy. Recall that GA_1 and GA_2 are lossless by design, since the fitness functions they use ensure that the selected subsets of features do represent the remaining features.

Table 11. Information Loss measures

Program	SU_1	SU_2	GA_1	GA_2	$H_{SU1 \cap GA2}$	$H_{SU1 \cup GA2}$
$print\_tokens2_{ALL}$	0	0	0	0	0.00378	0
tot_info_{ALL}	0.00305	0.00920	0	0	0.04237	0
$Schedule_{ALL}$	0	0.04910	0	0	0.00350	0
$space_{BB}$	0	0	0	0	0.00513	0
$Tomcat3.0_{ALL}$	0	0.00610	0	0	0.11601	0
$Tomcat3.2.1_{ALL}$	0	0.00833	0	0	0.01686	0
$Jigsaw_{ALL}$	0.00220	0	0	0	0.08179	0
$JTidy_{ALL}$	0.00146	0.02720	0	0	0.02001	0
$flex2_{BB}$	0	0	0	0	0.39411	0
$sed3_{BB}$	0	0	0	0	0.00578	0
$JTidy_{SliceP}$	0	0.02493	0	0	0.12874	0
Average	0.00066	0.01135	0	0	0.07437	0

D. Assessment 3: Quality of Cluster Analysis

We used four metrics to assess the impact of our reduction techniques on the quality of cluster analysis. First, we briefly describe them; then present and discuss their computed values. Two of the metrics we used are well known, namely, *V-Measure* and *F-Measure*; the other two, *M3-pass-fail* and *M3-multiple*, are our own.

1. V-Measure

V-measure [40] is a conditional entropy-based external cluster evaluation measure, which requires the knowledge of class labels for each data point in advance. It accounts for the homogeneity and completeness of clusters through their weighted harmonic mean.

$$\mathbf{V_{\beta} = \frac{(1+\beta)*h*c}{(\beta*h)+c}} \quad (22)$$

For a dataset of *N* points with a set of classes $C=\{c_1,.....,c_{|C|}\}$ and set of clusters $K=\{k_1,.....k_{|K|}\}$, *A* is a |C|x|K| contingency matrix representing the clustering solution such that $A = \{a_{ck}\}$ is the number of data elements that are members of class *c* and are assigned by the algorithm to cluster *k*.

Homogeneity is defined by having all the clusters containing only data points belonging to a single class. *H(C|K)=0* is obtained when each cluster contains only members of a single class; implying perfect homogeneous clustering. Note that *H(C)=0* occurs when there is only a single class and hence homogeneity is 1.

$$\mathbf{h = \begin{cases} 1 & \text{if } H(C) = 0 \\ 1 - \frac{H(C|K)}{H(C)} & \text{else} \end{cases}} \quad (23)$$

where:

$$\mathbf{H(C|K) = -\sum_{k=1}^{|K|}\sum_{c=1}^{|C|}\frac{a_{ck}}{N}\log\frac{a_{ck}}{\sum_{c=1}^{|C|}a_{ck}}} \quad (24)$$

$$\mathbf{H(C) = -\sum_{c=1}^{|C|}\frac{\sum_{k=1}^{|K|}a_{ck}}{N}\log\frac{\sum_{k=1}^{|K|}a_{ck}}{N}} \quad (25)$$

As for completeness, it is defined by having all the data points that are members of a given class to be elements of the same cluster. *H(K|C)=0* implies perfect completeness. Note that *H(K)=0* occurs when there is only a single cluster and hence completeness is 1.

$$\mathbf{c} = \begin{cases} \mathbf{1} & \mathbf{H(K) = 0} \\ \mathbf{1 - \frac{H(K|C)}{H(K)}} & \mathbf{else} \end{cases} \quad (26)$$

where:

$$\mathbf{H(K|C) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{a_{ck}}{N}\log\frac{a_{ck}}{\sum_{k=1}^{|K|}a_{ck}}} \quad (27)$$

$$\mathbf{H(K) = -\sum_{k=1}^{|K|}\frac{\sum_{c=1}^{|C|}a_{ck}}{N}\log\frac{\sum_{c=1}^{|C|}a_{ck}}{N}} \quad (28)$$

Note that the *V-Measure* values lie in [0, 1], where higher V values imply better clustering.

2. F-Measure

F-Measure is a mapping based external cluster evaluation measure [40], based on a post-processing step in which each cluster is mapped to a class. Different mapping schemes can lead to different quality scores for the same clustering.

For a dataset of *N* points with a set of classes $C=\{c_1,.....,c_{|C|}\}$ and set of clusters $K=\{k_1,.....k_{|K|}\}$, *A* is a $|C|x|K|$ contingency matrix representing the clustering solution such that $A = \{a_{ck}\}$ is the number of data elements that are members of class *c* and are assigned by the algorithm to cluster *k*.

$$\mathbf{F(C,K) = \sum_{c=1}^{|C|}\frac{|c|}{N}\max_{k\in|K|}\{F(c,k)\}} \quad (29)$$

$$\mathbf{F(c,k) = \frac{2*R(c,k)*P(c,k)}{R(c,k)+P(c,k)}} \quad (30)$$

$$\mathbf{R(c,k) = \frac{a_{ck}}{|c|}} \text{ and } \mathbf{P(c,k) = \frac{a_{ck}}{|k|}} \quad (31)$$

A higher value of *F*-Measure means a better clustering. The main drawback is that this measure is specific for the cluster-class matching [40]. When calculating the similarity between a hypothesized clustering and a “true” clustering, *F-Measure* only considers the contributions from those clusters that are matched to a target class. This is a problem as two different clustering can result in identical scores.

3. M3-pass-fail:

M3-pass-fail is our own external cluster evaluation metric developed earlier [10] with a minor modification. It considers the composition of the generated clusters in terms of failing and passing tests, assessing the extent of isolation of the failures. The first metric *M1* tries to answer the question *“Are failures isolated from passing tests?”*

$$\mathbf{M1} = \frac{\sum \mathbf{cluster\_score}}{\mathbf{n}} \text{ (32)}$$

where *cluster_score* = $|F - P|$; $F = \frac{f}{cluster\_size}$; $P = \frac{p}{cluster\_size}$. Here *f* is the number of failures in the cluster, *p* the number of passing tests in the cluster, *cluster_size* the number of tests in the cluster, and *n* is the number of clusters. The best case (*M1*=1.0) occurs when any given cluster contains all passing or all failing tests. The second metric answers the question *“Are failures clustered together?”*

$$\mathbf{M2} = \frac{\mathbf{1}}{\mathbf{NCF}} \text{ (33)}$$

where *NCF* is the number of clusters containing failures. The best case (*M2*=1.0) occurs when all failures reside in one cluster (which can also contain passing tests). The combined metric *M3* answers the question “*Are failures isolated from passing tests and are clustered together?”* by computing the weighted harmonic mean of *M1* and *M2*:

$$\mathbf{M3} = \left(\frac{(\boldsymbol{\beta}^2+\mathbf{1})*\mathbf{M1}*\mathbf{M2}}{\boldsymbol{\beta}^2*\mathbf{M1}+\mathbf{M2}}\right) \text{ (34)}$$

In our evaluation β was set to 1, to avoid weighting *M1* more than *M2* or vice versa.

4. M3-pass-bug:

M3-pass-bug is a modified version of *M3-pass-fail* where the composition of the generated clusters is no more considered in terms of failing and passing tests, but in terms of multiple bugs and passing tests, aiming to assess the extent of isolation of every single bug rather than the failures as a whole. *M1* is kept the same while *M2* is modified to answer the question *"Are similar bugs clustered together?"*

$$\mathbf{M2} = \frac{\mathbf{TNB}}{\sum_{\mathbf{i=1}}^{\mathbf{k}} \mathbf{bug\_count}} \quad (35)$$

where *TNB* is the total number of bugs, *k* is the number of clusters, and *bug_count* is the number of unique bugs encountered in every cluster. The best case (*M2*=1.0) occurs when each bug resides in one cluster (which can also contain passing tests). The combined metric *M3* answers the question "*Are failures isolated from passing tests and are the individual bugs clustered together?"* by using the same weighted harmonic formula as *M3-pass-fail* with a value of β =1.

In this part of the experiments, for each reduction technique and each subject program, we varied the number of clusters from 3 to 10 when computing the metrics. We performed our evaluation on small number of clusters so the homogeneity measure won`t be biased and thus the F-Measure. Moreover, ideally we seek that the clustering technique will cluster all test cases belonging to the same bug in a single cluster, similarly for the passing test cases. Since the number of bugs in each program does not exceed 7, so a max of 10 clusters is enough. To add, the assessment takes into consideration cluster number 3 and above since we already know that we are dealing with multi-faulted programs, hence at least 2 bugs must exist requiring at least 3 clusters (an additional cluster for the passing runs). Appendix B presents figures that show the four plots of the metrics for every subject program. Each plot compares the measure under study with the 5 reduction techniques and the no-reduction. The measure being assessed is the average measure obtained while varying the clusters from 3 to 10. Figure 1 presents the average of the *V-Measure* metric across the 11 subject programs. Similarly, Figure 2 for the *F-Measure*, Figure 3 for the *M3-pass-fail*, and Figure 4 for the *M3-pass-bug*.

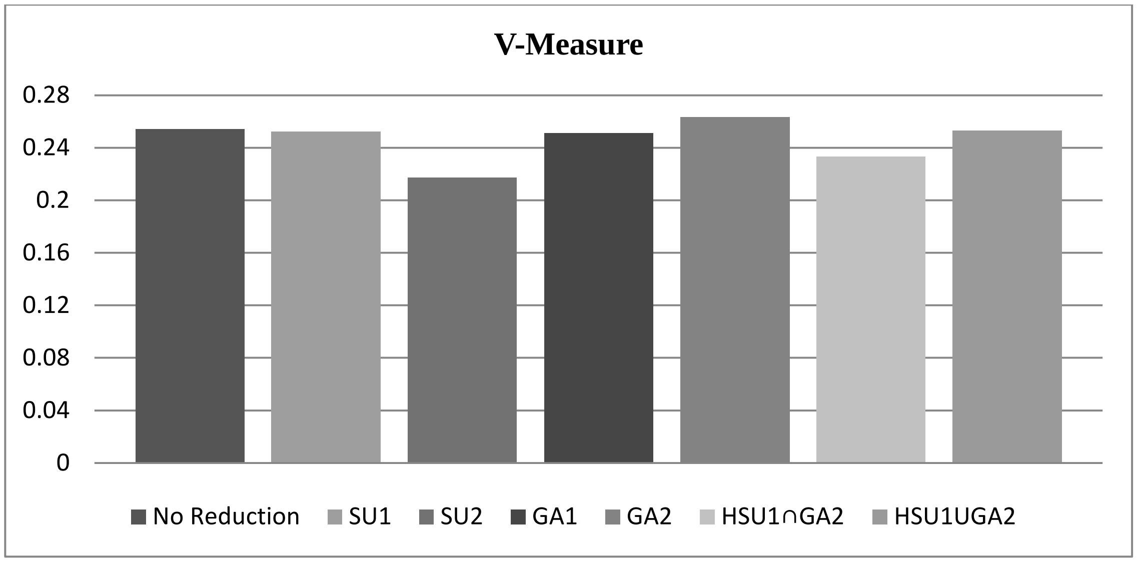


Figure 2. Average of the V-Measure Metric

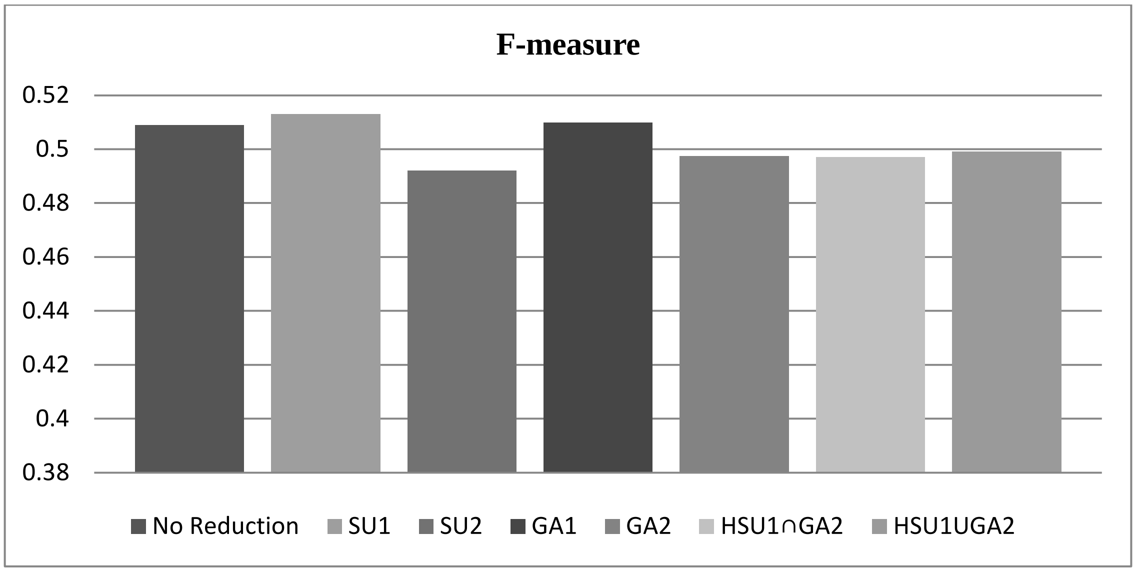


Figure 3. Average of the F-Measure Metric

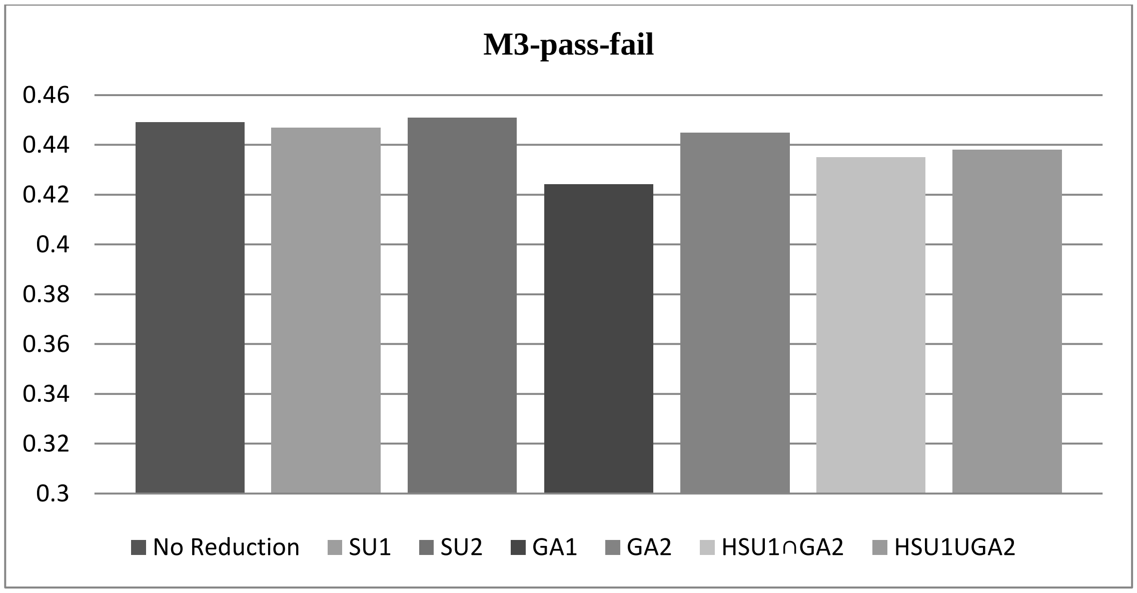


Figure 4. Average of the M3-pass-fail Metric

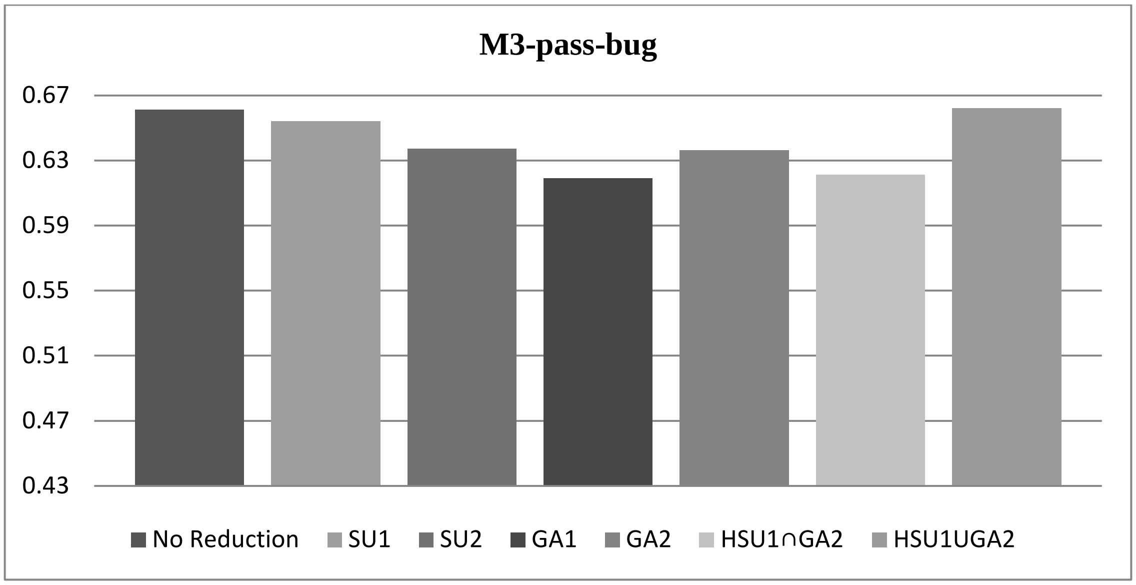


Figure 5. Average of the M3-pass-bug Metric

From the above four figures, it is not quite decisive which reduction technique has major positive enhancements on the cluster evaluation metrics. But it is noted that on average all the reduction techniques did not deteriorate the measures, and had a performance quite close to that with no reduction.

E. Assessment 4: Cost

Table 12 shows the times in seconds for each of the reduction techniques. Note that the times shown for $H_{SU1\cap GA2}$ and $H_{SU1\cup GA2}$ include the times for SU_1, GA_2, and the times for performing the intersection and unionizing of sets. The exhibited times are not very significant, except for the case of $JTidy_{SliceP}$ which involves a very large feature set.

Table 12. Cost of Reduction (in seconds)

Program	SU_1	SU_2	GA_1	GA_2	$H_{SU1\cap GA2}$	$H_{SU1\cup GA2}$
print_tokens2$_{ALL}$	15.04	28.95	21.35	23.87	40.33697	39.1061
tot_info$_{ALL}$	9.53	14.88	8.249	9.78	17.42377	17.4930
Schedule$_{ALL}$	43.01	67.48	88.78	91.56	134.8099	133.401
space$_{BB}$	299.19	623.01	327.86	383.56	680.0053	679.811
Tomcat3.0$_{ALL}$	60.52	92.43	44.54	45.93	83.11160	84.1543
Tomcat3.2.1$_{ALL}$	36.49	45.66	22.22	28.03	43.58441	44.1607
Jigsaw$_{ALL}$	54.11	88.65	40.77	43.37	73.21207	73.9935
JTidy$_{ALL}$	1679.527	3096.50	2187.66	2191.76	3844.019	3849.31
flex2$_{BB}$	31.79	65.99	9.78	9.93	36.61719	37.2851
sed3$_{BB}$	17.01	29.65	4.37	4.42	19.39623	19.9552
JTidy$_{SliceP}$	78733.13	174878.22	16971.1	17338.1	90685.83	90812.8

CHAPTER VI

IMPACT ON SOFTWARE ANALYSES

This chapter assesses our reduction techniques by studying the effect on two techniques: cluster-based test suite minimization and profile-based online intrusion and failure detection.

A. Tech-I: Cluster-based Test Suite Minimization

Test suite minimization involves selecting a subset of tests *T’* from an existing test suite *T* in order to reduce the cost of the testing process. An effective minimization technique would yield a *T’* that is manageable in size and that reveals all (or most of) the defects revealed by *T*. Coverage based test suite minimization techniques analyze the execution profiles of a program in order to construct a *T’* such that all the profiling elements covered by *T* are also covered by *T’* [34]. Distribution-based test suite minimization [34] techniques select test cases based on how their execution profiles are distributed in the multidimensional profile space. In this experiment, we use a distribution-based technique comprising the following steps:

- It applies *K-means* clustering to partition the population into *k* clusters. The Squared Euclidean distance measure is used as a dissimilarity metric, the k-initial centroids are chosen at random from the data set.
- One test is randomly selected from each cluster similar to what is done in [5]. This one-per-cluster sampling technique economically exercises each program behavior

represented by a cluster, and it also favors the selection of unusual executions, which tend to be placed in isolated clusters.

- The *k* selected tests represent the tests in the minimized test suite *T'*. *T'* is checked for the percentage of defects it covers. If 100 percent defect coverage is not achieved, the number of clusters is increased and the above steps are repeated.

For each program and reduction technique, Table 13 shows |*T'*| and the percent decrease in |*T'*| relative to *NoReduction*, denoted as *%Decrease*. For example, the minimized test suite following SU_1 that covers all the bugs in *print_tokens2* contains 250 test cases as opposed to 300 when no reduction is performed. In this case reduction had a positive impact on *Tech-I*, quantified as a 16.6% decrease in the number of test cases. Whereas, in the case of *Jigsaw*, reduction using SU_1 had a negative impact on *Tech-I*, as indicated by the negative value of *%Decrease*. We make the following observations based on the results shown in Table 13:

- In terms of average test suite minimization improvement ($\%Decrease_{avg}$), $H_{SU1 \cup GA2}$ performed best at 38%, followed by $H_{SU1 \cap GA2}$ at 25.8%, SU_1 at 25.6%, SU_2 at 21.5%, GA_2 at 9.5%, and then GA_1 at -2.9%.
- GA_2 is clearly the worst performer as it exhibited the lowest $\%Decrease_{avg}$ and the largest number of instances where *%Decrease* is negative.
- In the case of $Jigsaw_{ALL}$, all reduction techniques except for $H_{SU1 \cup GA2}$ had a negative impact on test suite minimization.

Table 13. Results for *Tech-I*

Program	*No Reduction*	SU_1	SU_2	GA_1	GA_2	$H_{SU1 \cap GA2}$	$H_{SU1 \cup GA2}$
print_tokens2$_{ALL}$	300	250	250	360	300	250	240
		16.6%	16.6%	-20%	0%	16.6%	20%
tot_info$_{ALL}$	920	900	700	902	870	610	700
		2.2%	23.9%	1.9%	5.4%	33.7%	23.9%
schedule$_{ALL}$	80	50	60	74	62	48	46
		37.5%	25%	7.5%	22.5%	40%	42.5%
space$_{BB}$	350	200	250	470	330	330	220
		42.8%	28.5%	-34.2%	5.7%	5.7%	37.1%
Tomcat3.0$_{ALL}$	150	60	80	100	74	84	52
		60%	46.6%	33.3%	50.6%	44%	65.3%
Tomcat3.2.1$_{ALL}$	100	100	80	120	100	80	78
		0%	20%	-20%	0%	20%	22%
Jigsaw$_{ALL}$	250	300	350	310	360	290	240
		-20%	-40%	-24%	-44%	-16%	4%
JTidy$_{ALL}$	60	40	50	60	60	48	40
		33.3%	16.6%	0%	0%	20%	33.3%
flex2$_{BB}$	90	50	60	90	88	70	42
		44.4%	33.3%	0%	2.2%	22.2%	53.3%
sed3$_{BB}$	90	50	40	78	46	34	30
		44.4%	55.5%	13.3%	48.8%	62.2%	66.6%
JTidy$_{SliceP}$	100	80	90	90	86	64	54
		20%	10%	10%	14%	36%	46%
%Decrease$_{max}$	-	60%	55.5%	33.3%	50.6%	62.2%	66.6%
%Decrease$_{avg}$	-	25.6%	21.5%	-2.9%	9.5%	25.8%	38%
%Decrease$_{min}$	-	-20%	-40%	-34.3%	-44%	-16%	4%

Next we provide a discussion that might shed some light on why most of the reduction techniques did not improve *Tech-I* for *Jigsaw* but $H_{SU1 \cup GA2}$ did. *Tech-I* is based on the premise that when a given fault is exercised, it induces a set of program elements (features) that closely characterizes it or correlates with it. Consequently, due to this set, the corresponding execution profile will be dissimilar from the others. It is possible that for a given fault(s) in *Jigsaw*, one subset of the elements of such set was retained by SU_1 and its complement retained by GA_2, so the full set could not be found unless the features of SU_1 and GA_2 were unionized, as it is the case in $H_{SU1 \cup GA2}$.

Finally, the costs in seconds for *Tech-I* are shown in Table 10. The table also shows the percent decrease in time relative to *NoReduction*, denoted as *%CostDecrease*. Note how the average *%CostDecrease* for all six reduction techniques was above 95%. To summarize, reduction was shown to have a considerable positive impact on both the effectiveness and efficiency of *Tech-I*.

Table 14. Cost of *Tech-I* (in seconds)

Program	*No Reduction*	SU_1	SU_2	GA_1	GA_2	$H_{SU1 \cap GA2}$	$H_{SU1 \cup GA2}$
print_tokens2$_{ALL}$	580.96	55.26	48.48	34.94	27.79	24.54	50.97
		90.4%	91.6%	93.9%	95.2%	95.7%	91.2%
tot_info$_{ALL}$	648.94	23.20	13.87	14.30	14.50	8.34	17.19
		96.4%	97.8%	97.7%	97.7%	98.7%	97.3%
schedule$_{ALL}$	238.77	21.49	20.21	23.84	18.10	11.98	16.91
		90.9%	91.5%	90.0%	92.4%	94.9%	92.9%
space$_{BB}$	1505.70	176.81	211.43	53.40	38.12	36.69	215.63
		88.2%	85.9%	96.4%	97.4%	97.5%	85.6%
Tomcat3.0$_{ALL}$	1323.17	3.95	5.08	2.15	2.67	2.53	3.91
		99.7%	99.6%	99.8%	99.7%	99.8%	99.7%
Tomcat3.2.1$_{ALL}$	564.45	3.02	1.92	1.62	1.82	1.17	2.43
		99.4%	99.6%	99.7%	99.6%	99.7%	99.5%
Jigsaw$_{ALL}$	2040.44	8.70	9.91	4.68	4.60	3.13	9.37
		99.5%	99.5%	99.7%	99.7%	99.8%	99.5%
JTidy$_{ALL}$	1000.72	45.56	37.50	4.25	3.89	3.17	48.38
		95.4%	96.2%	99.5%	99.6%	99.6%	95.1%
flex2$_{BB}$	73.91	2.20	3.10	1.57	1.65	1.25	2.34
		97.0%	95.7%	97.8%	97.7%	98.3%	96.8%
sed3$_{BB}$	10.03	1.07	0.87	0.72	0.52	0.53	0.72
		89.2%	91.2%	92.7%	94.7%	94.6%	92.8%
JTidy$_{SliceP}$	40472.58000	292.23	263.51	2.43	2.34	1.60	177.44
		99.2%	99.3%	99.9%	99.9%	99.9%	99.5%
%CostDecrease$_{avg}$	-	95.0%	95.3%	97.0%	97.6%	98.1%	95.4%

B. Tech-II: Profile-based Online Intrusion and Failure Detection

In [26], the authors proposed an intrusion/failure detection system (IDS) based on execution profiles. The proposed approach entails generating signatures that correlate with given attacks or failures, to be matched online during deployment. For each type of attack/failure, the IDS generates a corresponding signature using a training set containing both safe (or passing) and unsafe (or failing) tests. Such signature is in the form of a combination of program elements that highly correlates with the unsafe tests; i.e. it is executed by a high percentage of attacks (or failures) and a low percentage of safe (or passing) tests. Assuming that the execution profiles consist of *N* distinct elements (features), the search space would consist of 2^N combinations, which calls for the use of an approximation algorithm. Consequently, the IDS uses a genetic algorithm, which we will denote by IDS_GA, to carry out the search as follows:

- Each combination/solution/chromosome is represented by a bit string whose length is equal to *N*. A bit set to 1 (resp. 0) indicates that the corresponding element is included (resp. not included) in the combination
- The fitness of a combination *C* is evaluated as *%F(C)-%P(C)* where *%F(C)* (resp. *%P(C)*) represents the percentage of failures/attacks (resp. safe tests) exercising *C* in the training set. As such, the higher the fitness the better the combination is in terms of characterizing the attack.
- *IDS_GA* starts with an initial population that is entirely constructed from the intersection of the profiles of all attacks in the training set. This way, the *%F* component of the fitness function would be optimized.

- After creating the initial population, *IDS_GA* produces successive generations using crossover and replacement
  - *Crossover*: two (parent) combinations are randomly selected from the population and a child is generated as a combination containing program elements from both parents. Choosing these elements is done probabilistically, in a manner that favors the parent with the higher fitness. After evaluating the fitness of the child, it replaces the parent with the lesser fitness.
  - *Replacement*: a randomly-generated child replaces a randomly-chosen parent.
- The algorithm terminates when a solution with a fitness of 1.0 is encountered or when the maximum number of generations (MAX_GENERATIONS) is reached.
- Throughout the whole process, *IDS_GA* keeps track of the best encountered solution, which is returned at the end.

We now quantify the impact of reduction on the performance of *Tech-II*. Using the ten subject programs and corresponding 41 vulnerabilities/defects, we repeated the signature generation process using the reduced execution profiles. Table 15 shows the averages of the following entities: a) fitness of the resulting combinations; b) search time in milliseconds; c) size of the resulting combinations; d) percentage of false positives during online matching; and e) percentage of false negatives during online matching. Also note that in our experiments: a) we randomly selected training sets that are 10% the size of the full test suite; and b) to account for the non-determinism in *IDS_GA*, the values shown in the table are averages computed based on 10 runs. We make the following observations based on Table 15:

- SU_1 is the best performer in terms of *Fitness*, while $H_{SU1 \cup GA2}$ is a close second.
- SU_1 is the best performer in terms of *Time,* while $H_{SU1 \cup GA2}$ is a close second. Also, all reductions techniques performed better than *NoReduction* when it comes to *Time*.
- SU_1 is the best performer in terms of *%FP*, while $H_{SU1 \cup GA2}$ is a close second.
- All are somewhat comparable in terms of S*ize*. Noting that the size of the generated combination impacts the online matching efficiency of the *IDS*.
- All of the reductions techniques performed worse than *NoReduction* in terms of *%FN*. Noting that the performance of $H_{SU1 \cup GA2}$, SU_2, and SU_1 was not considerably worse.

In summary, $H_{SU1 \cup GA2}$ and SU_1 were: a) the better performers amongst the reduction techniques; b) more efficient than *NoReduction*; and c) little less effective than *NoReduction* due to their slightly higher *%FN*.

Table 15. Average results for Tech-II

	Fitness	*Time (ms)*	*Size*	*%FP*	*%FN*
NoReduction	0.984	5467.01	3.96	1.87%	9.27%
SU_1	0.998	3.20	3.95	0.47%	14.27%
SU_2	0.969	12.05	3.85	3.31%	13.76%
GA_1	0.937	8.81	3.91	6.37%	17.86%
GA_2	0.927	9.64	4.15	7.58%	18.83%
$H_{SU1 \cap GA2}$	0.910	9.95	3.88	9.12%	18.34%
$H_{SU1 \cup GA2}$	0.997	4.12	4.16	0.73%	13.72%

Even though the average search times for $H_{SU1 \cup GA2}$ and $SU_1$are no more than 0.07% of that of *NoReduction*, but that time improvement has no significant practical value given that the average search time for *NoReduction* is less than six seconds. With this in mind, we conducted an additional experiment using $JTidy_{SliceP}$ designed to likely benefit from the reduction of execution profiles. Specifically, we deliberately modified the classification of the profiles in a manner that no fitness of 1 would be found, which prevents *IDS_GA* from terminating before iterating MAX_GENERATIONS times. It should be noted that not finding a combination of fitness 1 is not unlikely, as it can be seen in Table 15. Table 16 shows the results for $JTidy_{SliceP}$ for that contrived experiment. As expected, no combinations with a good fitness were found, and the time for *NoReduction* is considerable, as it is hundreds of times larger than that of the rest.

Table 16. $JTidy_{SliceP}$ results for *Tech-II*

	Fitness	*Time (ms)*	*Size*	*%FP*	*%FN*
NoReduction	0.29	2534493.5	6.50	66.25%	28.35%
SU_1	0.30	17047.0	3.42	75.31%	18.48%
SU_2	0.30	11694.8	3.91	76.79%	17.62%
GA_1	0.29	164.0	2.0	68.75%	28.24%
GA_2	0.33	164.0	2.0	67.91%	24.53%
$H_{SU1 \cap GA2}$	0.21	111.5	1.5	81.67%	16.17%
$H_{SU1 \cup GA2}$	0.35	16027.6	4.0	72.93%	18.93%

Finally, one can argue that the time saving for generating a signature is insignificant compared to the time taken to reduce the execution profiles. But it should be noted that in *most cases* the reduction needs only to be done once with a given release, whereas the signature generation must be done every time a new attack is observed throughout the lifetime of a release. Actually, the reduction needs to be redone only in two cases: 1) when a newly discovered attack/failure covers some program elements that have not been previously covered; or 2) when the execution matrix resulting from adding the profile of a newly discovered attack/failure becomes inconsistent with the current reduced set. Note that the time cost for checking for both cases is negligible in practice.

CHAPTER VII

CONCLUSION AND FUTURE WORK

This work studied the impact of the size of execution profiles. It presented several reduction techniques and comparatively evaluated them by measuring the reduction rate, information loss, impact on the quality of cluster analysis, cost of reduction, and impact on two software analysis techniques. The results were promising as the average reduction rate ranged from 92% to 98%, most techniques were lossless or slightly lossy, the quality of cluster analysis slightly improved on average, and the cost of reduction was not very significant. The results also showed that reducing execution profiles can potentially benefit software analysis in terms of efficiency and/or effectiveness.

Given that with our proposed techniques we were able to achieve very high rates of reduction and very low rates of information loss with insignificant time cost, our future work should focus elsewhere on identifying other software analyses that majorly benefit from such reduction such as state profiling [29], and fault localization [1][23][25]. Furthermore, our experiments involved a limited number of subject programs; so further empirical studies are needed which involve a variety of other subject programs from different domains and environments.

REFERENCES

[1] Abou-Assi R. and Masri W. Identifying Failure-Correlated Dependence Chains. *First International Workshop on Testing and Debugging*, TeBug/ICST 2011, Berlin, March 2011.

[2] Abou-Assi R. and Masri W. Lossless Reduction of Execution Profiles using a Genetic Algorithm. Regression/ICST 2014, Cleveland, April 2014.

[3] Thomas Ball, James R. Larus: Efficient Path Profiling. MICRO 1996: 46-57

[4] Laura Maria Cannas, Nicoletta Dessì, and Barbara Pes. 2011. A hybrid model to favor the selection of high quality features in high dimensional domains. In *Proceedings of the 12th international conference on Intelligent data engineering and automated learning* (IDEAL'11), Hujun Yin, Wenjia Wang, and Victor Rayward-Smith (Eds.). Springer-Verlag, Berlin, Heidelberg, 228-235.

[5] W. Dickinson, D. Leon, and A. Podgurski, “Finding Failures by Cluster Analysis of Execution Profiles,” Proc. 23rd Int’l Conf. Software Eng., pp. 339-348, May 2001.

[6] W. Dickinson, D. Leon, and A. Podgurski, “Pursuing Failure: The Distribution of Program Failures in a Profile Space,” Proc. 10th European Software Eng. Conf./Ninth ACM SIGSOFT Symp. Foundations of Software Eng., pp. 246-255, Sept. 2001.

[7] R.O. Duda, P.E. Hart, and D.G. Stork. Pattern Classification. John Wiley & Sons, New York, 2edition, 2001.

[8] Joan Farjo, Rawad Abou Assi, Wes Masri, and Fadi Zaraket. Does Principal Component Analysis Improve Cluster-Based Analysis? Regression/ICST 2013, Luxembourg, March 2013.

[9] Joan Farjo and Wes Masri. Weighted Execution Profiles for Software Testing. Regression/ICST 2014, Cleveland, April 2014.

[10] Joan Farjo, Wes Masri, and Hazem Hajj. Isolating Failing Test Cases: A Comaprative Experimental Study of Clustering Technqiues. The 3rd Int`l Conference on Communications and Information Technology ICCIT 2013, June 2013, Lebanon

[11] Fodor I. K. A survey of dimension reduction techniques. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory. June 2002.

[12] Guyon,I. and Elisseeff,A. (2003) An introduction to variable and feature selection. J. Mach Learn Res., 3, 1157–1182.

[13] Quanquan Gu, Zhenhui Li, Jiawei Han, Generalized Fisher Score for Feature Selection, The 27th Conference on Uncertainty in Artificial Intelligence (UAI), Barcelona, Spain, 2011

[14] Maria Halkidi , Yannis Batistakis , Michalis Vazirgiannis, On Clustering Validation Techniques, Journal of Intelligent Information Systems, v.17 n.2-3, p.107-145, December 2001

[15] Jones J., Harrold M. J., and Stasko J. Visualization of Test Information to Assist Fault Localization. ICSE 2001,467-477.

[16] X. He, D. Cai, and P. Niyogi, “Laplacian Score for Feature Selection,” Proc. Advances in Neural Information Processing Systems, vol. 18, 2005.

[17] Hui-Huang Hsu; Cheng-Wei Hsieh; Ming-Da Lu, "A Hybrid Feature Selection Mechanism," Eighth International Conference on Intelligent Systems Design and Applications, 2008. ISDA '08., vol.2, no., pp.271,276, 26-28 Nov. 2008

[18] G.H. John, R. Kohavi, and K. Peger. Irrelevant feature and the subset selection problem. In W.W. Cohen and Hirsh H., editors, Machine Learning: Proceedings of the Eleventh International Conference, pages 121-129, New Brunswick, N.J., 1994. Rutgers University.

[19] I. Kononenko, “Estimating Attributes: Analysis and Extension of RELIEF,” Proc. European Conf. Machine Learning (ECML), 1994.

[20] H. Liu and R. Setiono. Chi2: Feature selection and discretization of numeric attributes. In J.F. Vassilopoulos, editor, Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence, November 5-8, 1995, pages 388-391, Herndon, Virginia, 1995. IEEE Computer Society.

[21] Josep M. Mateo-Sanz , Josep Domingo-Ferrer , Francesc Sebé, Probabilistic Information Loss Measures in Confidentiality Protection of Continuous Microdata, Data Mining and Knowledge Discovery, v.11 n.2, p.181-193, September 2005

[22] Masri, W. Exploiting the Empirical Characteristics of Program Dependences for Improved Forward Computation of Dynamic Slice. Empirical Software Engineering (ESE) (Springer), 2008 13:369-399.

[23] Masri, W. Fault Localization Based on Information Flow Coverage. Software Testing, Verification and Reliability (STVR) (Wiley), 2010, vol. 20(2), pp. 121-147.

[24] Masri W., Abou-Assi R. Cleansing Test Suites from Coincidental Correctness to Enhance Fault-Localization. Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April, 2010.

[25] Masri W. and Abou Assi R. Prevalence of Coincidental Correctness and Mitigation of its Impact on Fault-Localization. ACM Transactions on Software Engineering and Methodology (TOSEM). *Accepted*.

[26] Masri W., Abou Assi R, and El-Ghali M. Generating Profile-Based Signatures for Online Intrusion and Failure Detection. Information and Software Technology (IST) (Elsevier). Vol. 56, Issue 2, Feb. 2014, pages 238-251.

[27] Masri W., Abou-Assi R., El-Ghali M., and Fatairi N. An Empirical Study of the Factors that Reduce the Effectiveness of Coverage-based Fault Localization. International Workshop on Defects in Large Software Systems, DEFECTS, Chicago, IL, 2009.

[28] W. Masri, R. Abou Assi, F. Zaraket, and N. Fatairi. Enhancing Fault Localization via Multivariate Visualization, Regression/ICST 2012, Montreal, Canada, April 2012.

[29] Masri W., Daou J., and Abou-Assi R. and Masri W. State Profiling of Internal Variables. Regression/ICST 2014, Cleveland, April 2014

[30] Masri W., Halabi H. An algorithm for capturing variables dependences in test suites. Journal of Systems and Software (JSS) 84(7): 1171-1190 (2011).

[31] Masri, W. and Podgurski, A. Application-Based Anomaly Intrusion Detection with Dynamic Information Flow Analysis. Computers & Security (Elsevier). Vol. 27 (2008), pages 176-187.

[32] Masri, W. and Podgurski, A. Algorithms and Tool Support for Dynamic Information Flow Analysis. Information and Software Technology (IST) (Elsevier). Vol. 51 (Feb. 2009), pages 385-404.

[33] Masri, W. and Podgurski, A. Measuring the Strength of Information Flows in Programs. ACM Transactions on Software Engineering and Methodology (TOSEM). , Vol. 19, No. 2, Article 5, October 2009

[34] Masri W., Podgurski A. and Leon D. An Empirical Study of Test Case Filtering Techniques Based On Exercising Information Flows. IEEE Transactions on Software Engineering, July, 2007, vol. 33, number 7, page 454

[35] Narendra, Patrenahalli M.; Fukunaga, K., "A Branch and Bound Algorithm for Feature Subset Selection," Computers, IEEE Transactions on , vol.C-26, no.9, pp.917,922, Sept. 1977

[36] Michael Negnevitsky. Articial Intelligence: A Guide to Intelligent Systems. Addison-Wesley, 2005.

[37] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge,1988

[38] P. Pudil, J. Novovičová, and J. Kittler, “Floating search methods in feature selection,” Pattern Recognition Letters, Vol. 15, 1994, pp. 1119-1125

[39] Raileanu, L., Stoffel, K.: Theoretical comparison between the Gini index and information gain criteria. Annals of Mathematics

[40] Andrew Rosenberg, and Julia Hirschberg. Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning EMNLP-cOnll, PAGE 410—420.

[41] Y. Saeys, I. Inza, P. Larranaga, "A review of feature selection techniques in bioinformatics," Bioinformatics, vol. 23, p. 2507, 2007

[42] Shlens J. A Tutorial on Principal Component Analysis. Center for Neural Science, New York University. April 22, 2009.

[43] M.R. Sikonja and I. Kononenko, “Theoretical and Empirical Analysis of Relief and ReliefF,” Machine Learning, vol. 53, pp. 23- 69, 2003.

[44] L. Song, A. Smola, A. Gretton, J. Bedo, and K. Borgwardt, “Feature Selection via Dependence Maximization,” J. Machine Learning Research, 2007.

[45] Stearns, S.D. (1976). On selecting features for pattern classifiers.Third lnternat. Conf. on Pattern Recognition, Coronado, CA, 71-75.

[46] Z. Zhao and H. Liu, “Spectral Feature Selection for Supervised and Unsupervised Learning,” Proc. 24th Int’l Conf. Machine Learning (ICML), 2007.

[47] Zheng Zhao; Lei Wang; Huan Liu; Jieping Ye, "On Similarity Preserving Feature Selection," IEEE Transactions on Knowledge and Data Engineering, vol.25, no.3, pp.619,632, March 2013

[48] http://sir.unl.edu

APPENDIX A

Table 17. Data of *print_tokens2*$_{ALL}$

# of faults	7	# of passing testcases	1801
Initial # of test cases	4055	# of failing_fault1 testcases	205
# CC test cases	1706	# of failing_fault2 testcases	146
# of test cases after cleanup	2349	# of failing_fault3 testcases	19
% failure	23.32%	# of failing_fault4 testcases	20
		# of failing_fault5 testcases	96
		# of failing_fault6 testcases	33
		# of failing_fault7 testcases	29

Table 18. Data of *tot_info*$_{ALL}$

# of faults	6	# of passing testcases	791
Initial # of test cases	1052	# of failing_fault1 testcases	20
# CC test cases	113	# of failing_fault2 testcases	19
# of test cases after cleanup	939	# of failing_fault3 testcases	37
% failure	15.76%	# of failing_fault4 testcases	1
		# of failing_fault5 testcases	3
		# of failing_fault6 testcases	68

Table 19. Data of *Schedule*$_{ALL}$

# of faults	3	# of passing testcases	981
Initial # of test cases	2650	# of failing_fault1	1070

		testcases	
# CC test cases	355	# of failing_fault2 testcases	30
# of test cases after cleanup	2295	# of failing_fault3 testcases	214
% failure	57.25%		

Table 20. Data of *$Space_{BB}$*

# of faults	5	# of passing testcases	1900
Initial # of test cases	2000	# of failing_fault1 testcases	27
# CC test cases	0	# of failing_fault2 testcases	27
# of test cases after cleanup	2000	# of failing_fault3 testcases	11
% failure	5%	# of failing_fault4 testcases	9
		# of failing_fault5 testcases	26

Table 21. Data of *$Tomcat3.0_{ALL}$*

# of faults	4	# of passing testcases	460
Initial # of test cases	658	# of failing_fault1 testcases	150
# CC test cases	0	# of failing_fault2 testcases	38
# of test cases after cleanup	658	# of failing_fault3 testcases	6
% failure	30.09%	# of failing_fault4 testcases	4

Table 22. Data of *$Tomcat3.2.1_{ALL}$*

# of faults	3	# of passing testcases	473
Initial # of test cases	497	# of failing_fault1 testcases	18
# CC test cases	0	# of failing_fault2 testcases	2
# of test cases after cleanup	497	# of failing_fault3 testcases	4

% failure	4.82%		

Table 23. Data of *Jigsaw*$_{ALL}$

# of faults	4	# of passing testcases	490
Initial # of test cases	530	# of failing_fault1 testcases	1
# CC test cases	0	# of failing_fault2 testcases	2
# of test cases after cleanup	530	# of failing_fault3 testcases	4
% failure	7.547%	# of failing_fault4 testcases	33

Table 24. Data of *JTidy*$_{ALL}$

# of faults	3	# of passing testcases	820
Initial # of test cases	1000	# of failing_fault1 testcases	83
# CC test cases	0	# of failing_fault2 testcases	2
# of test cases after cleanup	1000	# of failing_fault3 testcases	95
% failure	18%		

Table 25. Data of *flex2*$_{BB}$

# of faults	3	# of passing testcases	376
Initial # of test cases	532	# of failing_fault1 testcases	5
# CC test cases	0	# of failing_fault2 testcases	132
# of test cases after cleanup	531	# of failing_fault3 testcases	18
% failure	29.19%		

Table 26. Data of *sed3*$_{BB}$

# of faults	3	# of passing testcases	161
Initial # of test cases	213	# of failing_fault1 testcases	3

# CC test cases	13	# of failing_fault2 testcases	18
# of test cases after cleanup	195	# of failing_fault3 testcases	13
% failure	17.43%		

Note: Initially sed3 has 4 bugs but due to removing testcases that are caused by more than one bug, no more test cases existed for a certain bug and hence the analysis was performed on 3 bugs.

Table 27. Data of *JTidy*$_{SliceP}$

# of faults	3	# of passing testcases	295
Initial # of test cases	475	# of failing_fault1 testcases	83
# CC test cases	0	# of failing_fault2 testcases	2
# of test cases after cleanup	475	# of failing_fault3 testcases	95
% failure	37.89%		

APPENDIX B

This Appendix presents for each subject program four plots of the four distinct cluster evaluation metric described in Chapter 4. Each plot compares the corresponding cluster evaluation measure with no-reduction against all the 6 reduction techniques. Note that the metric being assessed in each plot is the average measure obtained while varying the cluster count from 3 to 10.

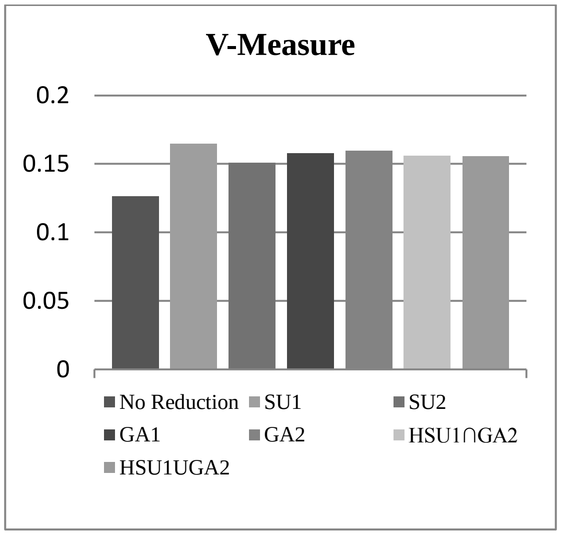


Figure 6. V-Measure *print_tokens2*$_{ALL}$

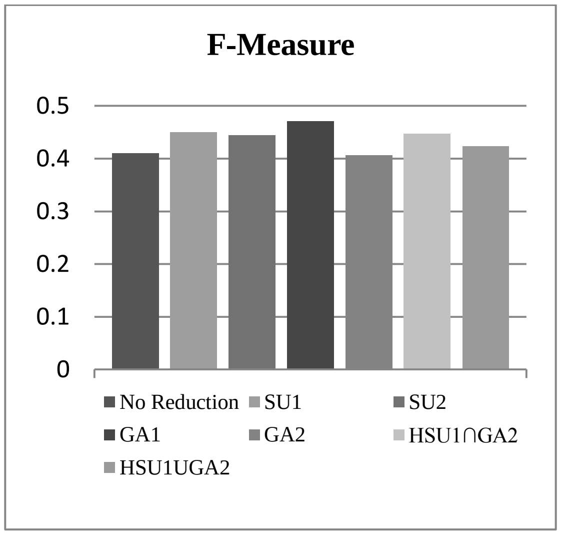


Figure 7. F-Measure *print_tokens2*$_{ALL}$

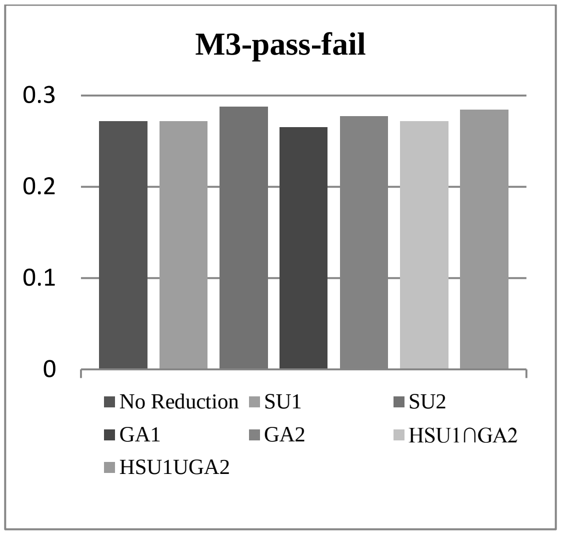


Figure 8. M3-pass-fail *print_tokens2*$_{ALL}$

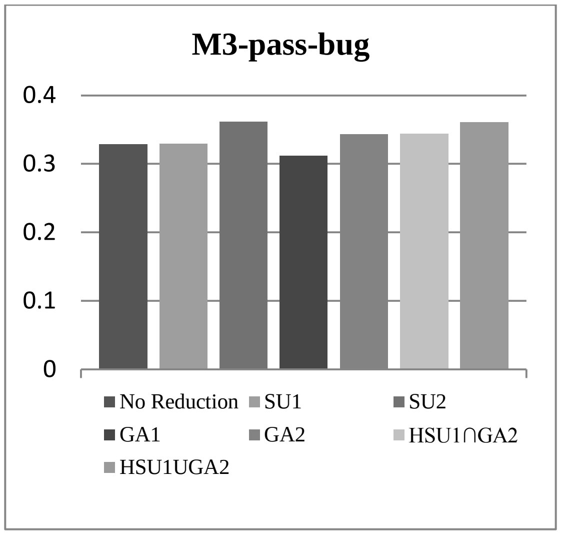


Figure 9. M3-pass-bug *print_tokens2*$_{ALL}$

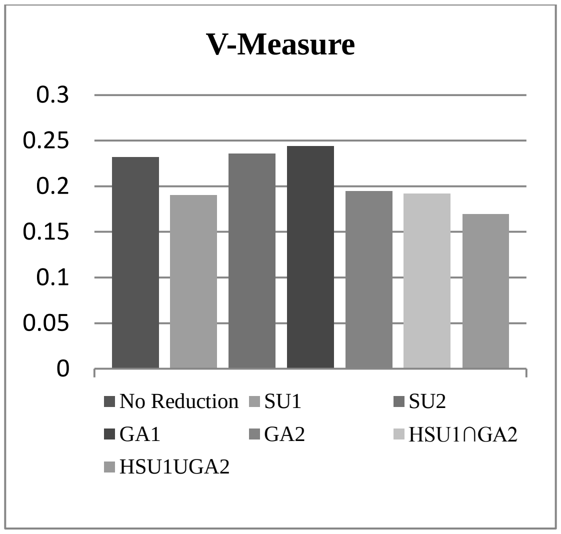


Figure 10. V-Measure *tot_info*$_{ALL}$

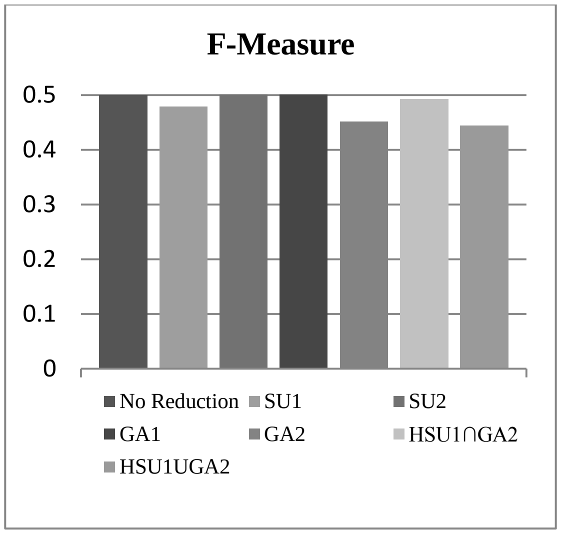


Figure 11. F-Measure *tot_info*$_{ALL}$

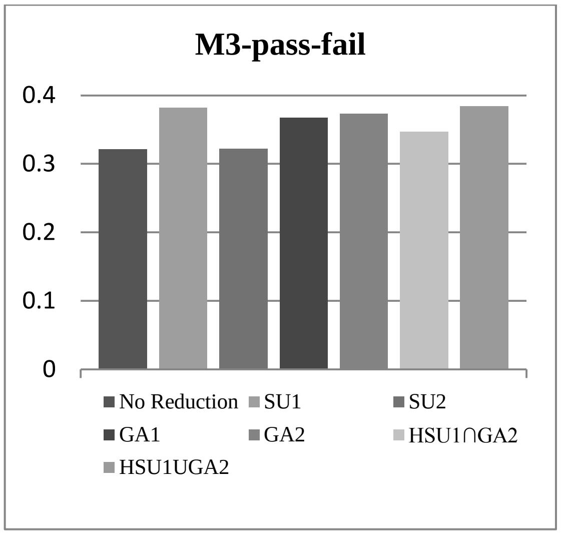


Figure 12. M3-pass-fail *tot_info*$_{ALL}$

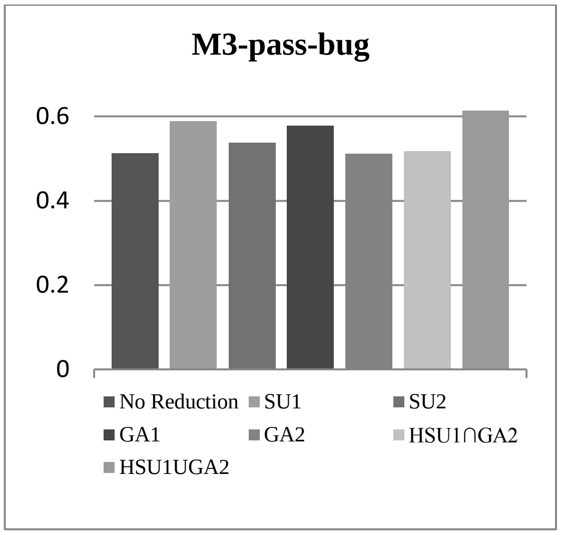


Figure 13. M3-pass-bug *tot_info*$_{ALL}$

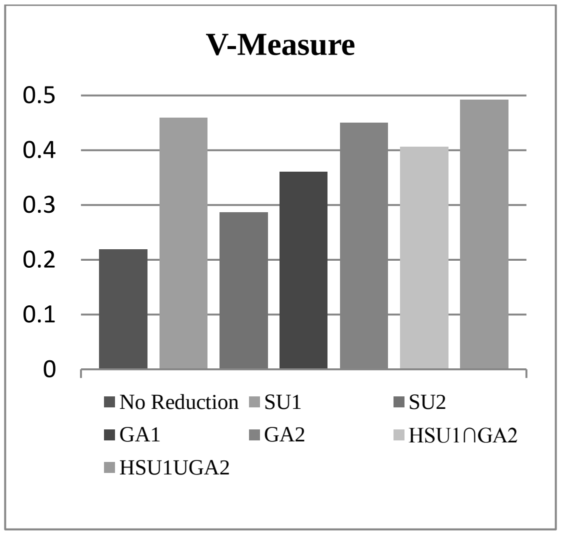


Figure 14. V-Measure *Schedule*$_{ALL}$

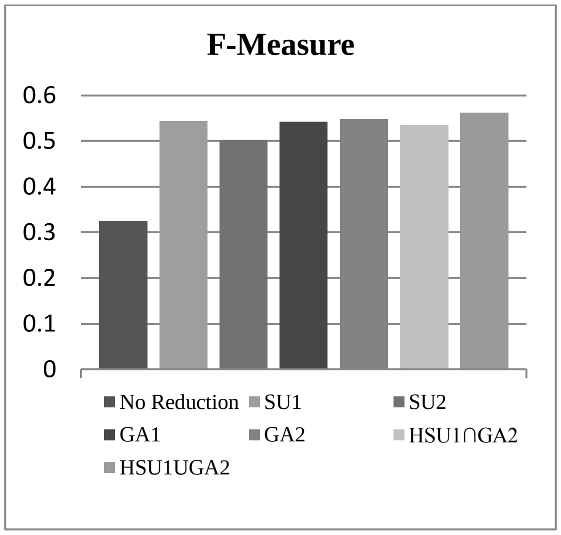


Figure 15. F-Measure *Schedule*$_{ALL}$

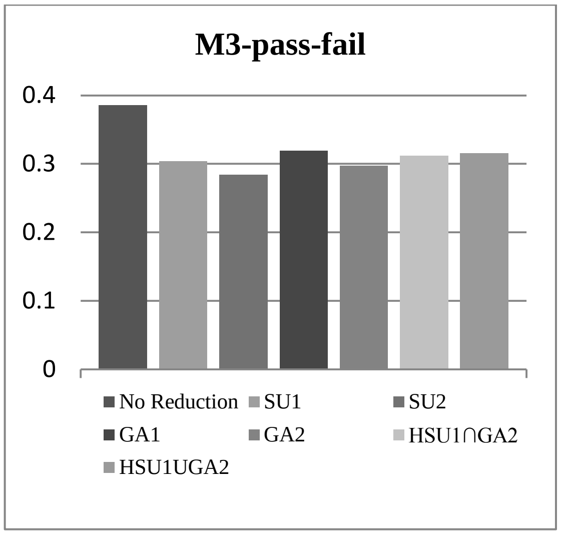


Figure 16. M3-pass-fail $\textit{Schedule}_{ALL}$

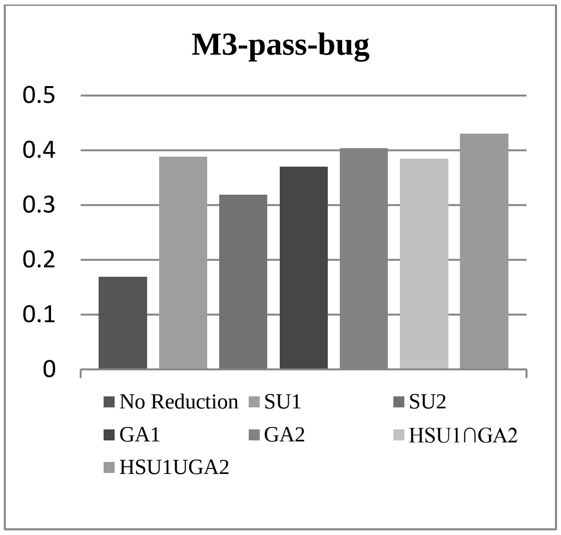


Figure 17. M3-pass-bug $\textit{Schedule}_{ALL}$

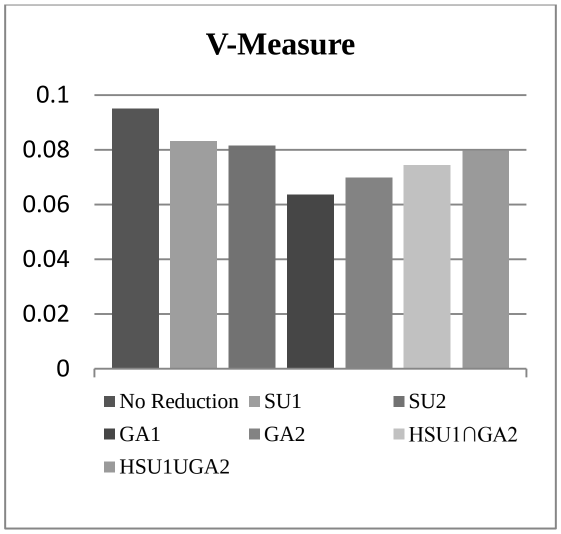


Figure 18. V-Measure $\textit{Space}_{BB}$

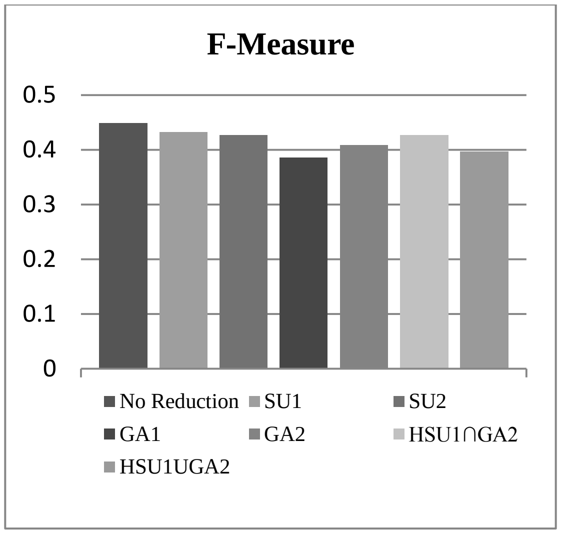
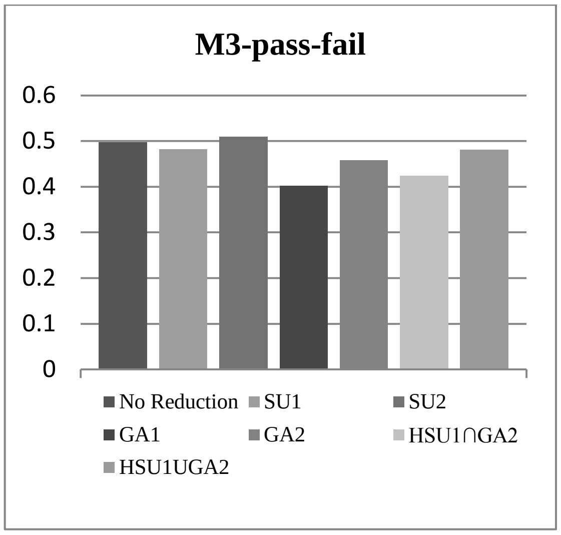


Figure 19. F-Measure $\textit{Space}_{BB}$



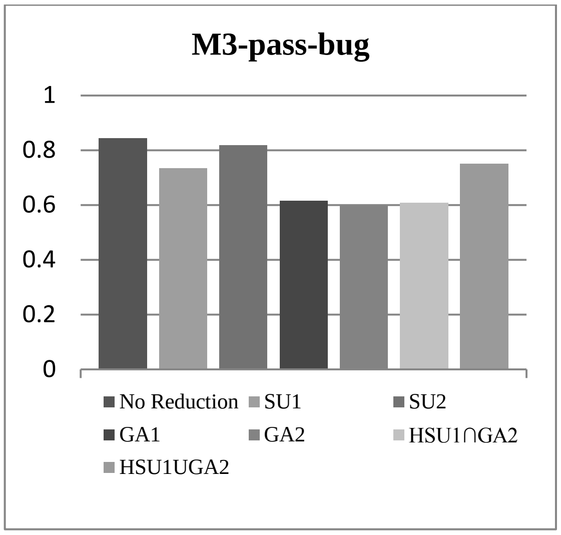
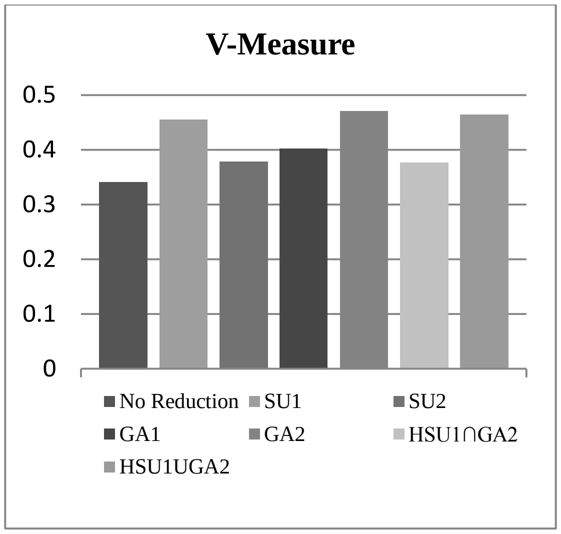


Figure 20. M3-pass-fail $Space_{BB}$

Figure 21. M3-pass-bug $Space_{BB}$



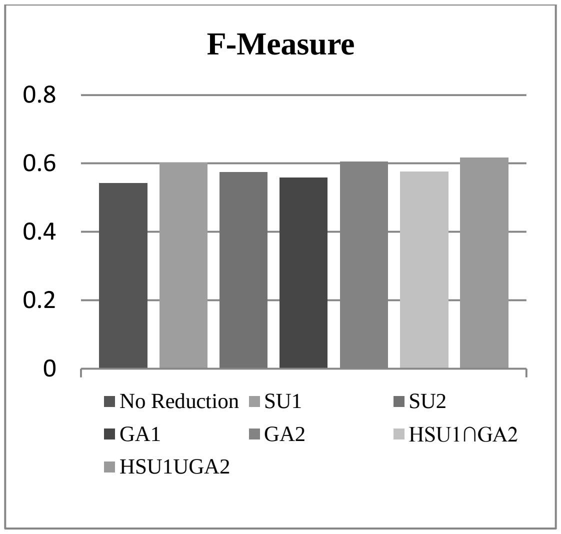


Figure 22. V-Measure $Tomcat3.0_{ALL}$

Figure 23. F-Measure $Tomcat3.0_{ALL}$

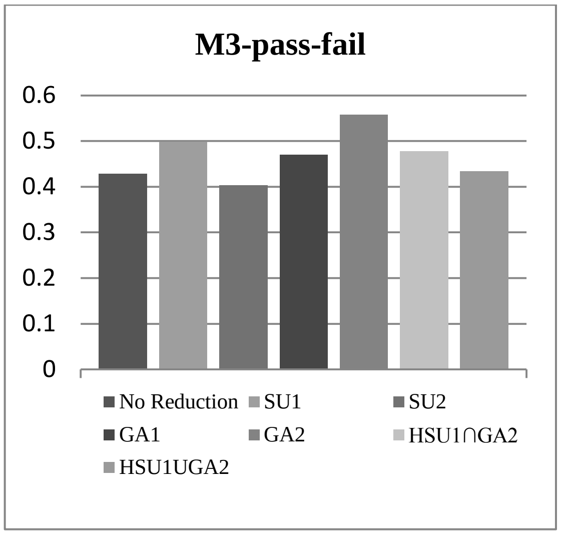


Figure 24. M3-pass-fail *Tomcat3.0*$_{ALL}$

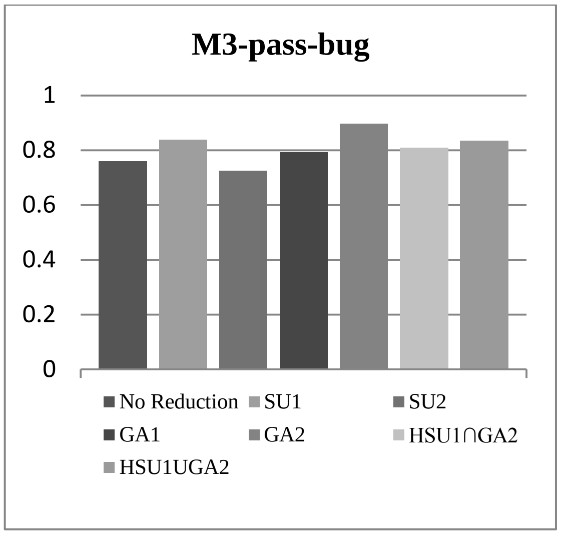


Figure 25. M3-pass-bug *Tomcat3.0*$_{ALL}$

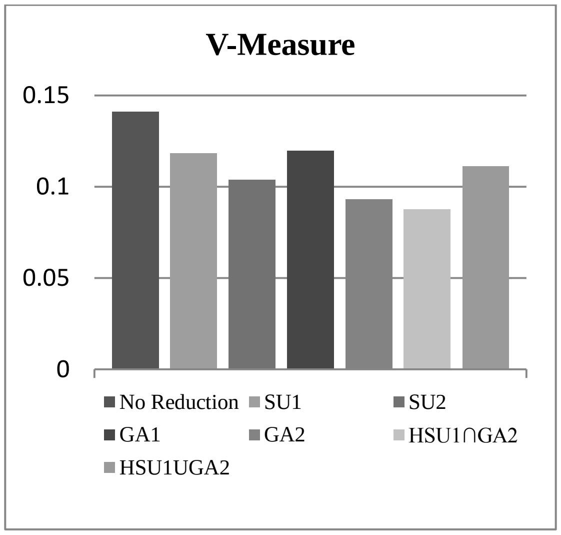


Figure 26. V-Measure *Tomcat3.2.1*$_{ALL}$

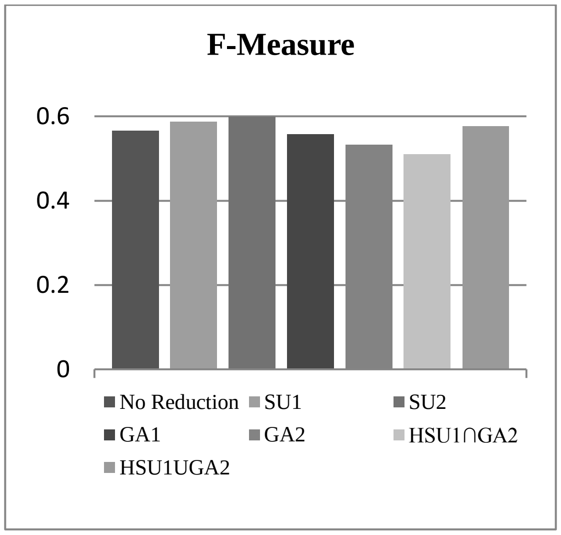


Figure 27. F-Measure *Tomcat3.2.1*$_{ALL}$

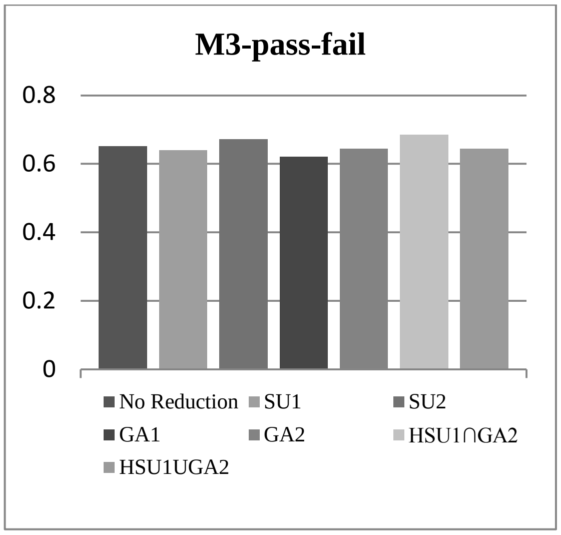


Figure 28. M3-pass-fail *Tomcat3.2.1*$_{ALL}$

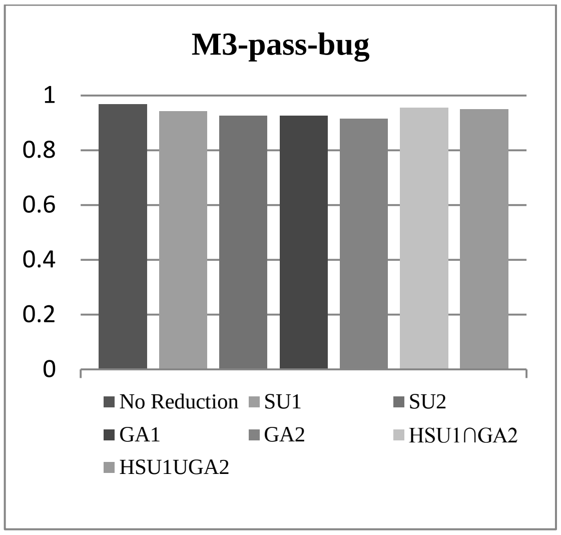


Figure 29. M3-pass-bug *Tomcat3.2.1*$_{ALL}$

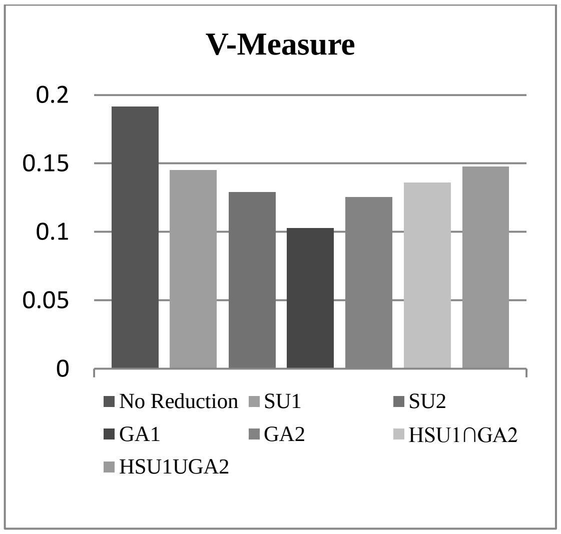


Figure 30. V-Measure *Jigsaw*$_{ALL}$

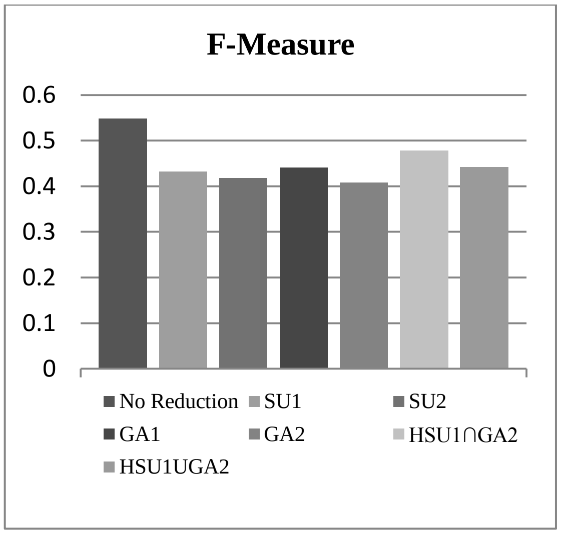


Figure 31. F-Measure *Jigsaw*$_{ALL}$

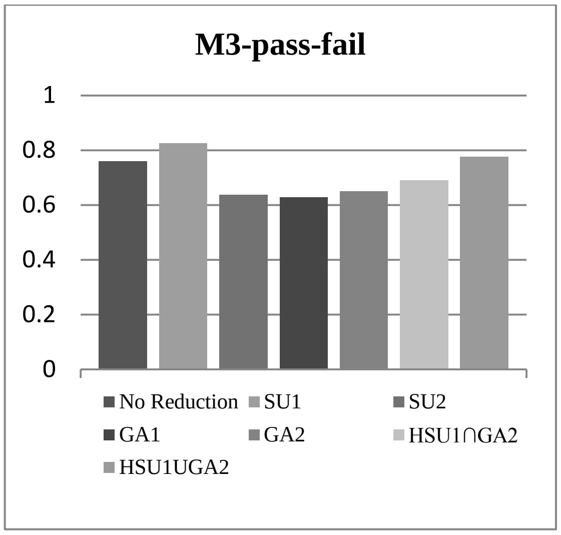


Figure 32. M3-pass-fail $Jigsaw_{ALL}$

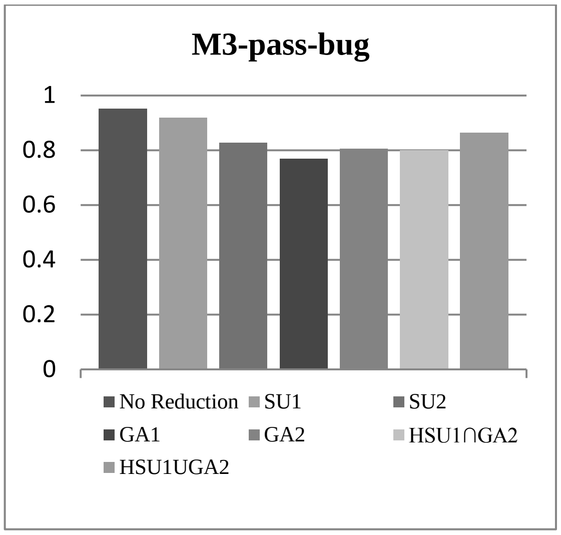


Figure 33. M3-pass-bug $Jigsaw_{ALL}$

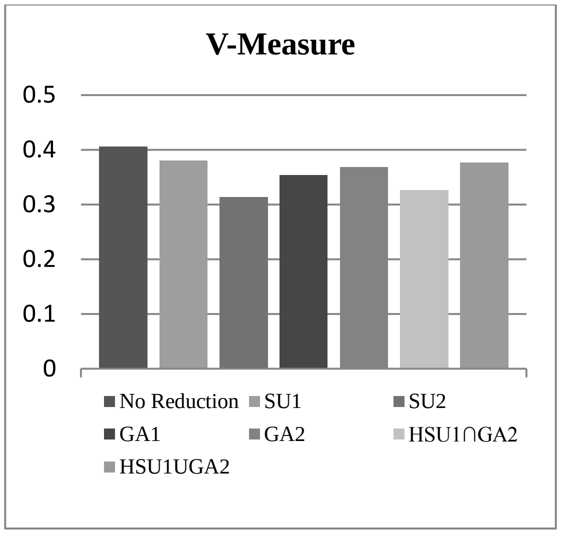


Figure 34. V-Measure $JTidy_{ALL}$

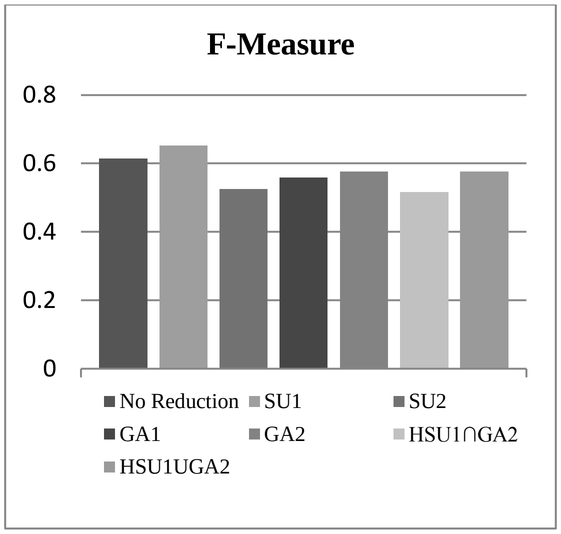


Figure 35. F-Measure $JTidy_{ALL}$

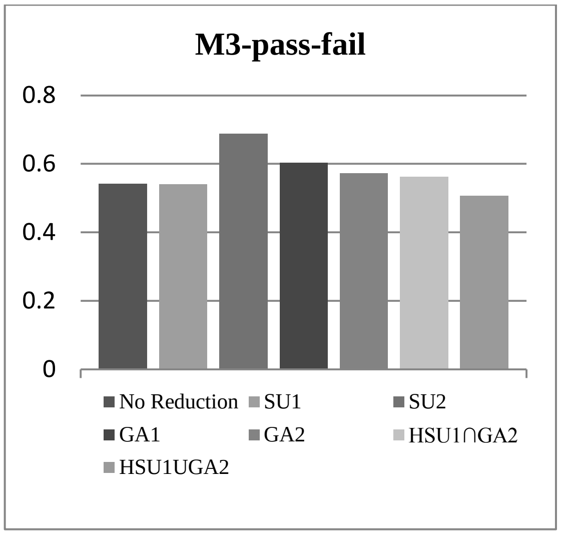


Figure 36. M3-pass-fail *JTidy*$_{ALL}$

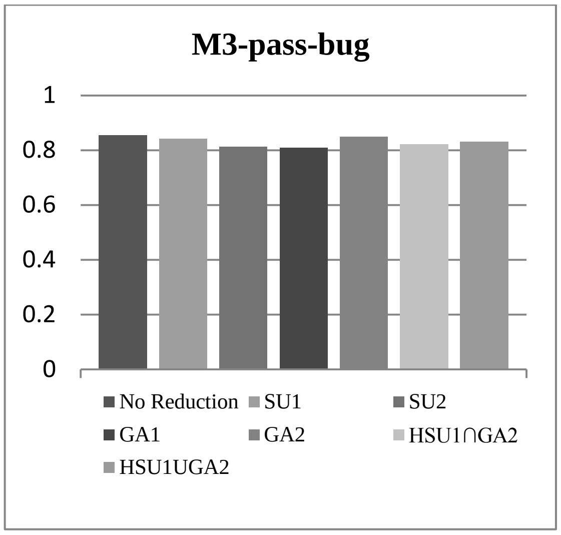


Figure 37. M3-pass-bug *JTidy*$_{ALL}$

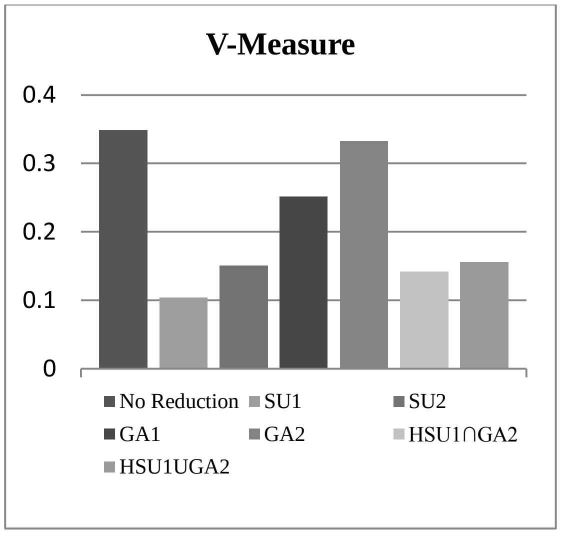


Figure 38. V-Measure *flex2*$_{BB}$

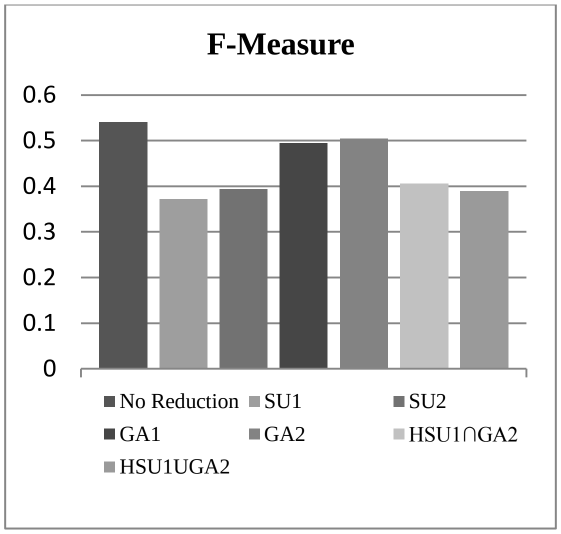


Figure 39. F-Measure *flex2*$_{BB}$

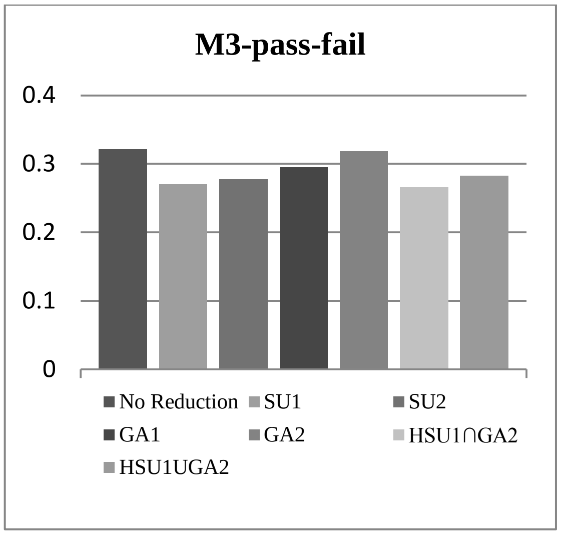


Figure 40. M3-pass-fail *flex2*$_{BB}$

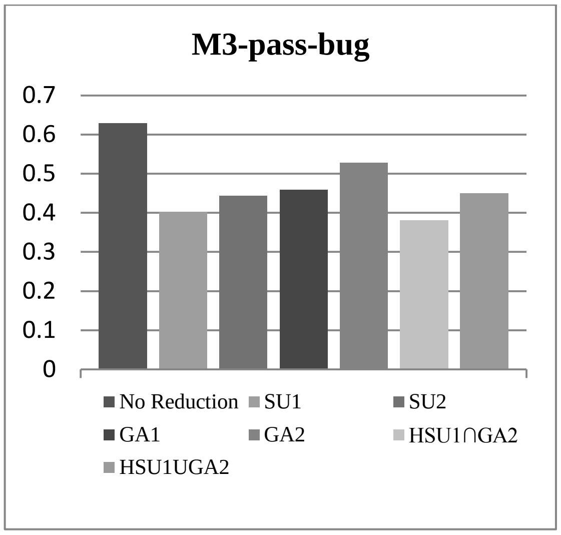


Figure 41. M3-pass-bug *flex2*$_{BB}$

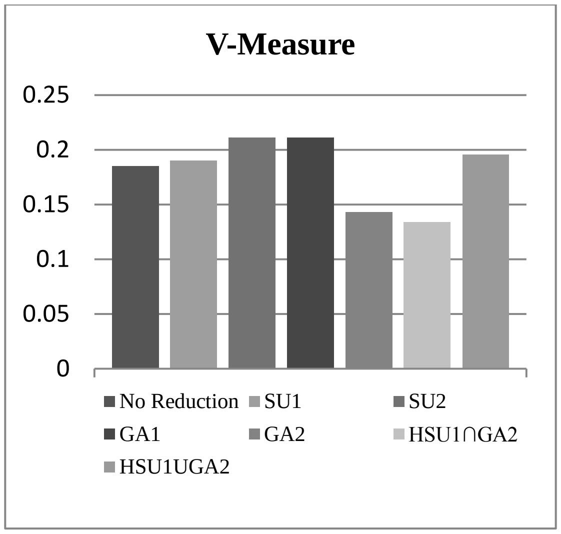


Figure 42. V-Measure *sed3*$_{BB}$

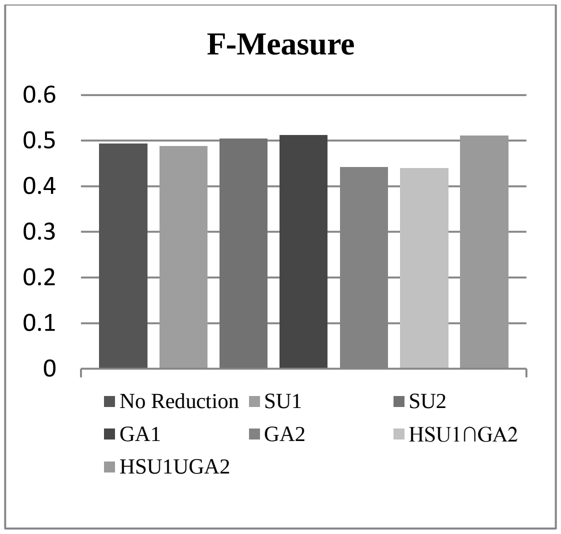


Figure 43. F-Measure *sed3*$_{BB}$

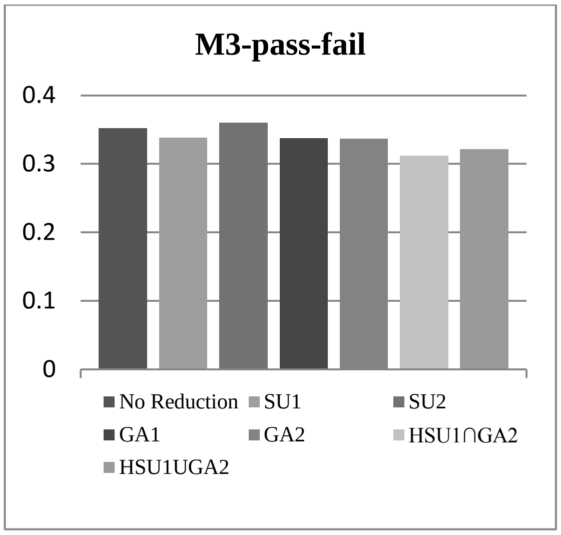


Figure 44. M3-pass-fail *sed3*$_{BB}$

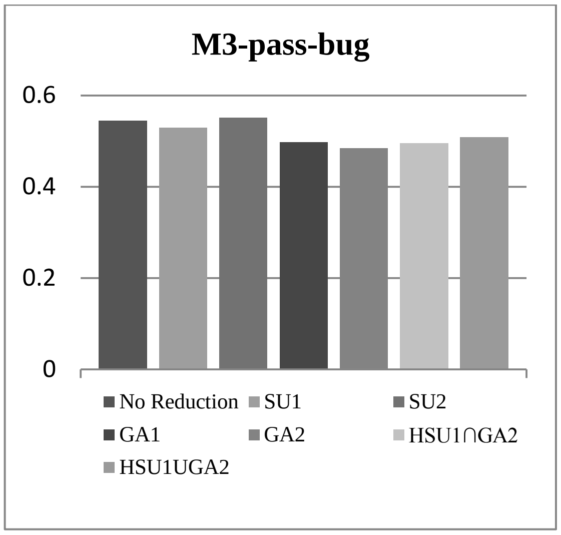


Figure 45. M3-pass-bug *sed3*$_{BB}$

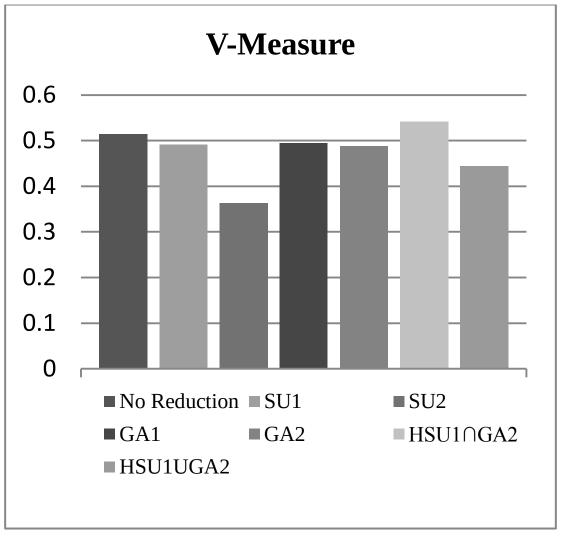


Figure 46. V-Measure *JTidy*$_{SliceP}$

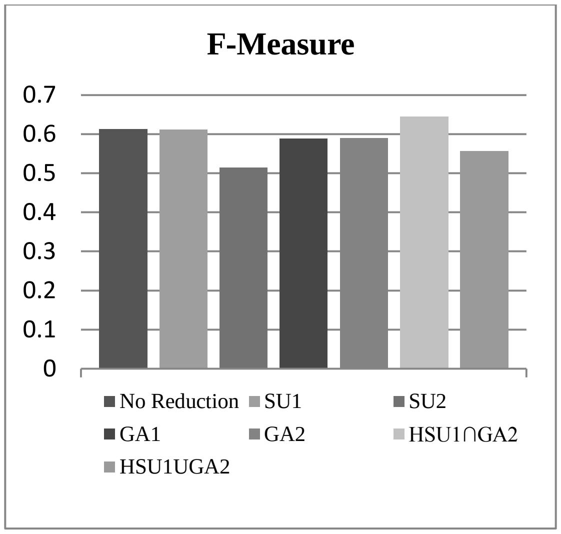


Figure 47. F-Measure *JTidy*$_{SliceP}$

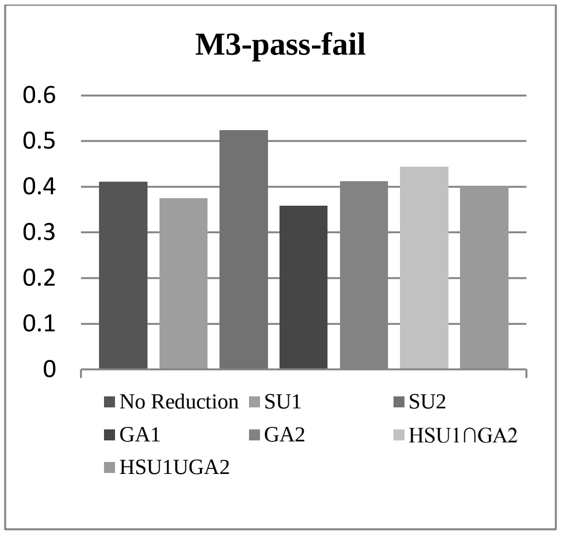


Figure 48. M3-pass-fail *JTidy*$_{SliceP}$

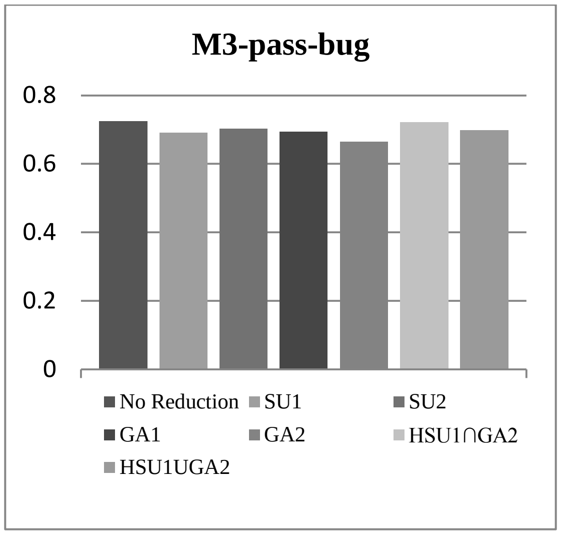


Figure 49. M3-pass-bug *JTidy*$_{SliceP}$