79

97-61

126137

N93-17268

Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development

Lionel C. Briand, Victor R. Basili and Christopher J. Hetmanski

Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, College Park, MD, 20742

To be published in the proceedings of the IEEE International Symposium on Software Reliability Engineering, North-Carolina, USA, October 1992.

Abstract:

Applying equal testing and verification effort to all parts of a software system is not very efficient, especially when resources are limited and scheduling is tight. Therefore, one needs to be able to differentiate low / high fault density components so that testing / verification effort can be concentrated where needed. Such a strategy is expected to detect more faults and thus improve the resulting reliability of the overall system. This paper presents an alternative approach for constructing such models that is intended to fulfill specific software engineering needs, (i.e. dealing with partial / incomplete information and creating models that are easy to interpret). Our approach to classification is to (1) measure the software system to be considered and (2) build multivariate stochastic models for prediction. We present experimental results obtained by classifying FORTRAN components developed at the NASA Goddard Space Flight Center into two fault density classes: low and high. Also, we evaluate the accuracy of the model and the insights it provides into the software process.

Key words: fault-prone software components, stochastic modeling, machine learning.

1. Introduction

In this paper, we address the issue of identifying high fault density software components via empirical stochastic modeling. If we can identify components that produce a great deal of faults relative to their size, then we can concentrate the verification and testing processes on them and thereby optimize the resulting reliability of the developed software system. However, building such stochastic models is a difficult task. The data collected is often incomplete and/or heterogeneous and presents many problems with respect to model construction (e.g. interdependencies, outliers, complex relationships). In this paper, we present an alternative modeling process based on both statistics and machine learning principles [M83]. We show how the process facilitates the identification of high fault density components based on metrics obtainable at the end of the coding phase.

The modeling approach presented in this paper, called Optimized Set Reduction (OSR), has been developed at the University of Maryland [BBT91] in the framework of the TAME project [BR88] . It is derived from the ID3 model [Q79, Q86, BR84] which was originally developed for automatic generation of classification/decision trees. As discussed in [CE87,BBT91], the use of ID3 has several inherent problems and leaves room for improvement with respect to many data analysis and modeling issues (i.e. small data sets, missing data values, noisy data, heteroscedasticity). Our motivation for developing OSR and a tool to support it was to design a data analysis technique matching, to the extent possible, the specific needs of building multivariate empirical models for software engineering. The issue of using OSR for predicting on a continuous range is addressed in [BBT91]. In this paper, we discuss using OSR to classify software components into two fault density classes (low, high).

In Section 2, we present the basic principles of the OSR algorithm and formally define the approach. This formalism is intended to give an unambiguous presentation of some of the features of OSR rather than a complete definition of it. Section 3 discusses the issue of building models based on partial information (i.e. missing data for technical or cost reasons). Section 4 presents a process called "pattern merging" whose goal is to facilitate interpretation and learning based on the generated models. Sections 5 and 6 present some of the results obtained via

Research this study was supported in part by NASA grant NSG 5123 and by AFOSR 90-0031

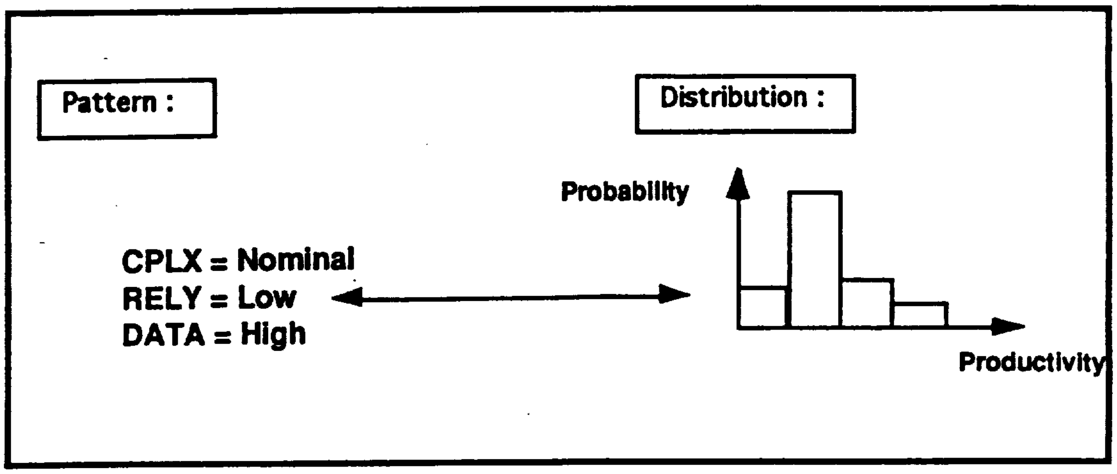

Figure 1: Example of a Pattern and its Associated Probability Distribution

experimentation using OSR. Based on these results, we can determine the accuracy of the model. Also, we can compare OSR's outputs with those of a *logistic regression* based model, which is one of the most standard statistical techniques for classification [HL89, AG90]. Finally, Section 7 underlines the major conclusions and directions for future research.

2. Optimized Set Reduction

2.1 Basic Principles

Let us assume we want to assess a particular characteristic of an object (e.g. the fault density of a component). We will refer to this characteristic as the Dependent Variable (Y). The object is represented by a set of explanatory variables which describe the software component (called Xs). These variables can be either discrete or continuous. For example, a software component may be described by two Xs, its cyclomatic complexity (continuous) and the type of its function (discrete). Also, assume we have a historical data set containing a set of *pattern vectors* that contain the previously cited Xs plus an associated actual Y value. We will call the Xs portion of the pattern vector a *measurement vector.*

The goal of the OSR algorithm is to determine which subsets of experiences (i.e. pattern vectors) from the historical data set provide the best characterizations of the object to be assessed. In other words, we try to determine which subsets of the data set yield the "best" probability distributions on the Y range. A good probability distribution is a probability distribution concentrating a large number of pattern vectors in either a small part of the range (Y is continuous) or in a small number of dependent variable categories (Y is discrete). One of the commonly used probability distribution evaluation functions is the *information theory entropy* (H). Alternative probability distribution evaluation functions are discussed in [Q86, SP88, M89]. Each of the subsets of the historical data set yielding "optimal" distributions, referred to as *optimal subsets,* are characterized by a set of conditions (referred to as *predicates*) which are true for all pattern vectors in that subset. Each set of predicates characterizing a subset is called a *pattern.* Figure 1 shows an example of a pattern and its associated probability distribution in the data set. The pattern is composed of three predicates where the dependent variable to be assessed is "development productivity". Figure 1 shows that if these predicates (i.e. ComPLeXity = Nominal, RELiabilitY=Low, DATA base size = High) are true for a project, then its productivity is most likely to be in the second productivity class.

2.2 Formal Definition of the OSR Process

We want to identify *optimal* subsets in the historical data set. We can formalize the process using set theory and predicate calculus by defining the function Opt. Let us assume we have a set of m explanatory variables $\{X_1, X_2, \dots, X_m\}$ and a corresponding set of explanatory variable value domains $\{EV_1, EV_2, \dots, EV_m\}$. Let us define the measurement vector domain to be $MV = \underset{i=1..m}{\times} EV_i$. The dependent variable value domain (DV) may be seen as a set of classes which can be either intervals or categories. Therefore, the value domain of the pattern vectors in the data set can be represented as $PV = DV \times MV$. Let PVS be a set of pattern vectors representing the historical data set ($PVS \subseteq PV$). A predicate is a variable value pair (i.e. an X_i and its corresponding explanatory variable value).

• Definition 1: Let PSS be a subset of PVS and let the measurement vector mv describe the object to assessed. VALID(PSS, mv) is true if mv is composed by at least one predicate which is true for all the pattern vectors in the set PSS.

$$\mathrm{PSS} \subseteq \mathrm{PVS} \wedge mv \in \mathrm{MV} \wedge \exists i \in \{1 .. m\}$$
$$\text{such that } \forall pv \in \mathrm{PSS}\ (mv(i) = pv(i))$$
$$\Rightarrow \mathrm{VALID}(\mathrm{PSS}, mv)$$

• Definition 2: TC(PSS, PVS) is true if the two data sets PSS and PVS do not show a statistically significant difference in distribution on the DV range. This is may be evaluated by performing statistical inference tests for comparing distributions. We currently use a binomial test for proportions since it does not have any applicable restraints (e.g. minimum expected frequencies like the Chi-squared test of independence)[CA88]. For each dependent variable class, the probability that proportions in PSS and PVS differ by chance is calculated. If for at least one of the classes, this probability is below a level of significance TC defined by the user, then we reject the hypothesis that the two distributions are identical. TC stands for *Termination Criterion* because the OSR process will be terminated if the condition defined by TC is true.

• Definition 3: EMIN(PSS_1, PVS) is true if PSS_1 is one of the subsets of PVS yielding a minimal ***normalized entropy*** H upon all statistically significant subsets of pattern vectors (e.g. a one vector subset has a minimal entropy but it is not a statistically significant subset and therefore is not relevant here).

$$\left(\mathrm{PSS}_1 \subseteq \mathrm{PVS} \wedge \neg\, \mathrm{TC}(\mathrm{PSS}_1, \mathrm{PVS})\right)$$
$$\wedge \left(\forall \mathrm{PSS}_2 \subseteq \mathrm{PVS}\ (\neg\, \mathrm{TC}(\mathrm{PSS}_2, \mathrm{PVS}) \wedge H(\mathrm{PSS}_1) \leq H(\mathrm{PSS}_2))\right)$$
$$\Rightarrow \mathrm{EMIN}(\mathrm{PSS}_1)$$

where
$$H(\mathrm{PSS}) = \sum_{d \in DV} - p(\mathrm{PSS}, d) \log_{|DV|} p(\mathrm{PSS}, d)$$
where
p(PSS, d) is the a priori probability that a vector which is an element of PSS has a dependent variable value belonging to the dependent variable class d

• Definition 4: Opt(PVS, mv) is a function yielding a set of *optimal pattern vector subsets.*

$$\mathrm{Opt}(\mathrm{PVS}, mv) = \{\mathrm{PSS} \subseteq \mathrm{PVS} \mid \mathrm{VALID}(\mathrm{PSS}, mv) \wedge \mathrm{EMIN}(\mathrm{PSS}, \mathrm{PVS})\}$$

However, the function Opt as defined cannot be used as an algorithm to extract the optimal subsets. The most important reasons are:

•The number of possible predicate combinations makes the search execution time prohibitive.
•We want the patterns to contain a minimal set of predicates, i.e., we want all the predicates in the pattern to have a significant impact on the resulting pattern entropy.
•We loose some information about the relative impact of the various predicates in the entropy reduction process.
•The contexts in which the various predicates appear relevant are undetermined.

Therefore, we implement a *greedy* algorithm using the function Opt which addresses the issues mentioned above. The Optimized Set Reduction algorithm can be roughly described by a three step recursive algorithm.

• Step 1: If the dependent variable is continuous, its range is divided into a set of classes according to two main factors: the necessary model accuracy and the size of the data set. Then, the ranges / categories of the explanatory variables are divided / clustered into *classes* (e.g. $Class_{i1}$ ... $Class_{ij}$ for the explanatory variable X_i) based on meaningful class creation techniques. For example, a Complexity range can be divided in three classes: low, average, high. Numerous techniques can be used in order to create meaningful classes (e.g. cluster analysis) [DG84]. However, this issue will not be addressed in this paper.

• Step 2: Select all the pattern vectors in the data set having a value for the explanatory variable X_i belonging to $Class_{ik}$, where the X_i for the object to be assessed belongs to the same class, and where the subset characterized by the predicate $X_i \in Class_{ik}$ yields the minimum statistically significant value for H. However, several subsets (characterized by different predicates) yielding "similar" minimal entropies (i.e. the similarity criterion has to be defined by the user of the algorithm) can be extracted at once. Let us call PSS_i the extracted subsets of pattern vectors.

• Step 3: Step 2 is repeated in a recursive manner on each subset PSS_i and each successive subset until the user defined termination criteria (TC) is reached.

This OSR algorithm can be formally specified as a two parameter recursive function where PVS is the historical data set and mv the vector describing the object to be assessed:

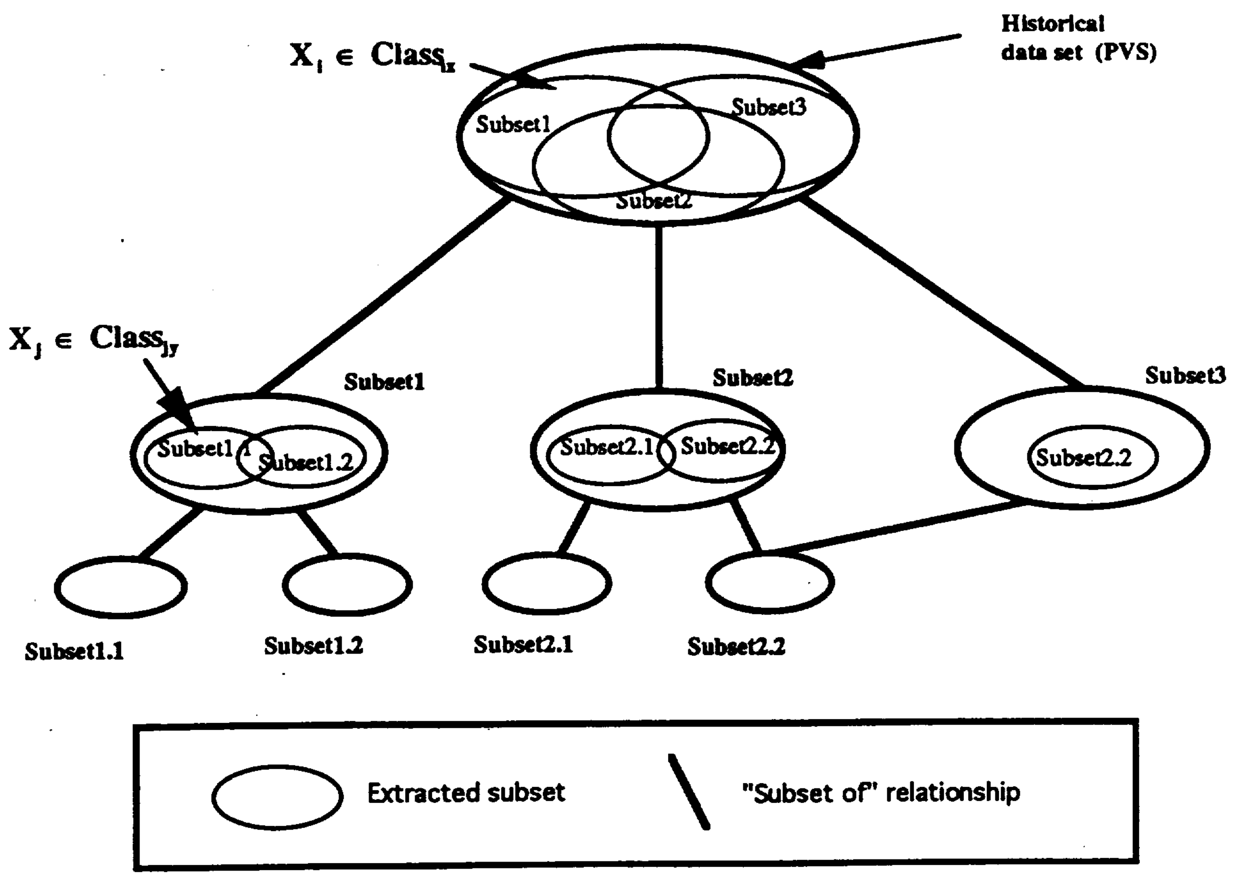

Figure 2: Example of OSR Hierarchy

$$\mathrm{OSR(PVS, mv)} = \text{if } \mathrm{Opt(PVS, mv)} \neq \varnothing \text{ then } \bigcup_{\mathrm{PSS} \in \mathrm{Opt(PVS, mv)}} (\mathrm{OSR(PSS, mv)}) \text{ else } \{\mathrm{PVS}\}$$

The whole subset extraction process can be represented as a hierarchy (see Figure 2). Note that this representation should not be confused with a partition tree since: (1) the extracted subsets are not exclusive and (2) a subset can have several parent subsets. Each path of the hierarchy represents a generated pattern (e.g. Figure 2: $X_i \in Class_{ix}$ AND $X_j \in Class_{jy}$ defines Subset1.1) which is relevant to the particular prediction to be performed. As shown in Figure 2, two patterns may yield exactly the same subset (e.g. Subset 2.2). The extracted subsets (i.e. leaves of the hierarchy) which form various probability distributions across the dependent variable range may show different trends. For each leaf probability distribution, if the dependent variable is discrete, the dependent variable class containing the largest number of pattern vectors may be selected as the most likely class for the new object (characterized by mv) to lie in. Using an alternative *Bayesian approach*, a *loss / risk function* could be defined by the user [BBT91]. In this case, the dependent variable class yielding the minimum *expected loss* is selected. Each pattern prediction (i.e. hierarchy leaf) is used to make a final global prediction based on predefined decision rules. In order to perform such decisions effectively, we need to be able to evaluate the accuracy of the identified patterns. This issue is treated in Section 3.

3. Handling Partial Information with OSR

3.1 Definition of the Problem

As mentioned above, analyzing complex data sets and variable relationships is a very difficult task for several reasons (i.e. incomplete / heterogeneous / small data sets, missing data, complex interdependencies). The most common of these is the problem of partial information. Our lack of understanding of software processes (due to our lack of experience and the wide variability from one development environment to another) makes experience difficult to reuse. Also, because of cost and schedule related constraints, necessary data cannot always be collected. All of these issues contribute to the incompleteness of our data.

Missing information reduces our ability to predict and understand. However, we have to establish whether or not the lack of a piece of data is an obstacle to prediction. This means that we need a model that both generates predictions and provides some insight into the reliability of each individual prediction. A goodness indication at the model level such as the coefficient of determination in least-squares regression analysis is not sufficient since it fails to yield an individual reliability measure for each prediction.

For example, let us say we wish to predict project productivity according to collected physical features of the system and predefined quality requirements. Suppose we do not have any information about the team experience related to the programming environment and the application domain. This information might be somewhat irrelevant, i.e. if the structural complexity of the software and the required system reliability are low, then the variance of the prediction is small. However, if high reliability on a complex software system is expected, then people rated as having low experience are likely to generate schedule and/or budget slippages. This will make any prediction based exclusively on other criteria meaningless. Therefore, we need a modeling approach that can answer the question: Do I have enough information to make a reliable prediction?

3.2 Solutions to Partial Information within the OSR Framework

For each measurement vector in the historical data set we run the OSR algorithm using as an initial data set (i.e. set at the top of the OSR hierarchy) the historical data set minus the measurement vector to be predicted. It is removed from the data set in order to avoid any bias in the results. We therefore extract specific patterns for each measurement vector and form a set of patterns representing the trends observable on this particular data set.

This resulting set of patterns, or Specific Pattern Set (SPS) may be seen as a model of the historical data set. Many of these patterns will be the same or "similar" and will therefore form classes of patterns. For each of these classes, based on the SPS, we can evaluate statistics such as *pattern reliability* (i.e. percentage of correct classification) or *pattern significance* (i.e. the probability that the reliability is greater than or equal to the one observed by chance) by comparing the predicted DV values with the actual ones. These statistics can then be used to evaluate predictions as explained in the subsequent paragraphs. The process of generating a SPS will be referred as to *Development Environment Analysis* (DEA).

In the text below, we assume the produced patterns have the following conjunctive normal form:

Predicate1 AND Predicate2 AND ... AND PredicateN

However, a pattern is not only a logical proposition. The order in which the predicates appear in the hierarchy (Figure 2) is relevant from an understanding perspective. A predicate is relevant only when the conditions defined by its preceding / parent predicates in the hierarchy (i.e. referred as to the *context* of a predicate in a particular pattern) are true. For example, Predicate 1 significantly reduces entropy by itself. Also, in the context of Predicate1, Predicate2 significantly reduces entropy. However, based on this pattern, there is no evidence that Predicate2 significantly reduces entropy by itself.

The notion of pattern reliability and significance, as mentioned above, can be more formally defined as follows: the reliability of a pattern with respect to a particular dependent variable class is the probability that the pattern will predict the correct value for the dependent variable.

Let $DVclass_i$ be dependent variable class i. Let T equal the number of generated patterns $\{P_j\}$ that predict $DVclass_i$. Let C equal the number of patterns which correctly predict $DVclass_i$ (based on the actual DV value of the pattern vector for which the pattern was produced during DEA).

Then we define the *reliability* of P_j with respect to the dependent variable class $DVclass_i$ as:

$$R\ [DVclass_i\ ;\ Pj] = C / T$$

The probability that a pattern appears T times yielding a particular classification $DVclass_i$ C times correctly by chance (P(C,T,p)) can be expressed by the binomial distribution:

$$P(C,T,p)=\frac{T\,!}{C\ !(T-C)!}p^{c}(1-p)^{T-C}$$

where, $p = p(DVclass_i)$, i.e. the a priori probability that the value of the dependent variable is in $DVclass_i$.

If the pattern reliability R is equal to 1.0, then the binomial equation can be simplified and the level of significance is simply p^T. If R is below one, then the *pattern significance* S can be calculated by using the following formula:

$$S=\sum_{i=0}^{T-C}P(C+i;T;p)$$

Since we are able to differentiate *significant, reliable* patterns from the non-significant and/or unreliable ones, we can assess the reliability of the prediction when we make it. A prediction based on a reliable pattern with a sufficient level of significance (e.g. $S < 0.05$) is believable, whereas, one based on a reliable pattern with a poor level of significance is not. A poor reliability means that a pattern is not robust to "noise" (i.e. the dependent variable variations created by unknown or non-measured explanatory variables). A poor significance may mean that the pattern is a result of noise or more complex phenomena which are beyond the scope of this paper.

4. A Process for Merging Patterns

Patterns are useful both for predicting variables of interest (e.g. fault density) and providing understandable / interpretable models. However, interpreting the patterns generated by a DEA would force the user to deal with useless complexity. Many of these patterns are similar and should not be differentiated. This can prevent the user from getting a clear picture of the model trends. Therefore, the patterns generated by the OSR process need to be grouped in order to make them more easily understandable and interpretable. This can be done using a formally defined statistical process (described below) where the user fixes the desired level of "similarity" between pattern by assigning values to a small set of parameters.

Let us define two patterns PT1 and PT2:

PT1: $X_j \in \text{Class}_{jy}$ AND $X_i \in \text{Class}_{ix}$
PT2: $X_j \in \text{Class}_{jy}$ AND $X_k \in \text{Class}_{kt}$

Suppose in the context where $X_j \in \text{Class}_{jy}$, the pattern vector set for which $X_i \in \text{Class}_{ix}$ happens to show a strong *association* with the one for which $X_k \in \text{Class}_{kt}$. This implies that these predicates capture basically the same phenomenon. The strength of the association can be assessed by using normalized Chi-squared based statistic such as Pearson's Phi [CA88]. A Chi-squared test can be performed in order to assess the statistical level of significance of such an association. The two patterns will be merged into one signifying that the selection of one predicate, or the other, during the OSR process, occurs by random. This is a result of slight differences between the two predicates and therefore distinguishing between them does not help to understand the object of study. This phenomenon is mainly due to complex interdependencies between Xs that are often underlying the software engineering data sets.

The notion of a "slight difference" is rather subjective and therefore must be defined by the user. Thus, he / she declares either a Phi value (actually Phi ^{2} which better represents in this case the degree of association [CAP88]) or a level of significance which represents the minimal degree of association necessary to assume two predicates as similar. This process of *merging patterns* based on the *similar predicates principle* yields the resulting pattern PT{1,2} which contains the *composite predicate* ($X_i \in \text{Class}_{ik}$ OR $X_k \in \text{Class}_{kt}$), implicitly meaning that its two component predicates are interchangeable in this context.

PT{1,2}: $X_j \in \text{Class}_{jy}$ AND ($X_i \in \text{Class}_{ix}$ OR $X_k \in \text{Class}_{kt}$)

Automated merging of similar patterns can be performed if the user provides either a Phi value or a level of significance that would correspond to an unambiguous definition of *pattern similarity*.

In a similar manner, we can define a second merging principle. Let us suppose we have the following patterns:

PT1: $X_j \in \text{Class}_{jy}$ AND $X_i \in \text{Class}_{ix}$
PT2: $X_j \in \text{Class}_{jy}$ AND $X_i \in \text{Class}_{it}$

Let us assume that Class_{ix} is a neighbor class of Class_{it} on the X_i range. In this particular case, if the two patterns characterize subsets with no statistically significant difference in distribution on the DV range, then they can be merged. This is because the variation from one class to the other seems to have a non-relevant effect on the dependent variable in the context where $X_j \in \text{Class}_{jy}$. Therefore, in order to assess if merging is possible, the probability that differences between distributions are due to random is calculated. For each dependent variable class, the proportions of pattern vectors are compared between the two distributions by calculating the probability that difference in proportion is due to random. If for all dependent variable classes, the resulting minimum probability is above a user-defined critical probability value, we accept the hypothesis that there is no significant difference between the two distributions. In the current tool, this is calculated through a binomial test in order to avoid the assumptions related to other more computationally effective tests (e.g. Chi-squared test of independence) [CAP88].

Both of the merging principles defined above can be used simultaneously in order to obtain more general patterns. However, the merging process using both of them must be carefully defined. In a tool, such mechanisms can be completely automated. The user would have to define some thresholds / criteria allowing the algorithm to declare two predicates *similar* (i.e., a level of significance, Phi value) and/or two classes *similar* (i.e., critical probability value). Before the merging process starts, the tool will calculate the matrix containing all the phi values and levels of significance between all predicates. Then, the merging process for the first position predicates starts: it is a several pass process where only two predicates can be merged at a time. First, predicates are merged according to the *similar class principle*. Then, the pairs of predicates showing the strongest significant associations are merged (*similar predicate principle*). During the next passes, predicates can be merged to composite predicates and composite predicates to composite predicates. The process stops when no merging is possible according to the criteria defined by the user. Once finished, association matrices are calculated within the contexts defined by each unique first position predicate (composite or not) resulting from the first pass. Then, the merging process for second position predicates begins within each context following the rules defined above. This is repeated successively on increasing predicate positions until a predefined (i.e. by the user) *maximum merging level* is reached. Thus, the user defines the number of predicate positions he / she wants to look at and therefore set the maximum merging depth of the algorithm.

5. Experiment Design

Our goal in this article is to describe a technique to distinguish between low and high risk components.

The notion of risk has multiple dimensions. We focus here on the identification of low/high fault density components. If we can distinguish between these two types of components, then we can concentrate on the high fault density ones during the verification and testing process. Moreover, if we can build this kind of model for each kind of fault, we can apply fault specific testing techniques to localize and correct faults. Basili and Selby showed in [BS87] that the effectiveness of three of the most well known testing approaches could vary significantly according to the type of fault considered. Although more experiments are needed to better understand the issue, this study supports the idea of building different models for each type of fault.

The collected data set is based on fifteen FORTRAN projects which were developed at the NASA Goddard Space Flight Center in the early eighties. On all of these project, static measures at the component level were collected using a static code analyzer. Fault report forms were filled out during the test phases of the development process. Faults were identified, classified according to a predefined taxonomy and localized in the system.

Our definition of fault density is the ratio of the number of faults over the number of executable statements. In this experiment we will look, as a first step, to faults related to incorrect data structure readings or writings (called "data value" faults in the NASA Software Engineering Laboratory). This type of fault represents about 50 percent of the total number of faults collected on the projects studied in this experiment.

6. Experimental Results

6.1 Prediction Results

We used the OSR technique to build classification models that were intended to provide an answer to the question: Is this component likely to be in the lowest / highest quartiles on the "data value" fault density range? This was done by performing a DEA on the data set which contained 399 pattern vectors. Each pattern vector was comprised of a list of static measures which describe a software component (i.e. the measurement vector), plus, the fault density of that component. Thereby, we were able to calculate an average classification correctness (i.e. percentage of components correctly classified) of the OSR model . Also, we try to demonstrate through examples that reliable patterns can be differentiated from misleading patterns.

For the sake of simplicity, we will look only at the two first predicates (the most relevant according to the OSR selection mechanism) of each of the generated patterns. R, O and S are respectively the Reliability, number of Occurrence (the number of times a pattern appeared), and the Significance of the pattern. The explanatory variable ranges were divided into quartiles. This method is the simplest technique for class creation but most likely the least effective. The class creation process is one of the issues that remains to be investigated (See Conclusion). OSR suggested that low and high fault density components were partly characterized by the following significant (< 0.05 level of significance) and non-significant patterns:

Low Fault Density Components

Assume that Fq, Sq, Tq and Lq represent respectively the First quartile, Second quartile and so forth, on the explanatory variable ranges.

• Examples of Highly-Significant Reliable Patterns:
PT1: # stmts $\in$ Lq AND # calls $\in$ Fq,
R = 1.0, O = 18, S = 0.000
PT2: # stmts $\in$ Lq AND # calls $\in$ Sq,
R = 1.0, O = 17, S = 0.000
PT3: # stmts $\in$ Lq AND # format/stmt $\in$ Fq,
R = 1.0, O = 10, S = 0.000
PT4: # stmts $\in$ Lq AND # i/o stmt / stmt $\in$ Fq,
R = 1.0, O = 15, S = 0.000
PT5: # stmts $\in$ Lq AND # assign/stmt $\in$ Fq,
R = 1.0, O = 8, S = 0.004
PT6: # stmts $\in$ Lq AND # decis_node/stmt $\in$ Fq,
R = 1.0, O = 11, S = 0.005
PT7: # stmts $\in$ Lq AND #funct/stmt $\in$ Tq
R = 1.0, O = 24, S = 0.000

PT8: # decision nodes $\in$ Lq AND # calls $\in$ Fq,
R = 1.0, O = 14, S = 0.000
PT9: # decision nodes $\in$ Lq AND # calls $\in$ Sq,
R = 1.0, O = 15, S = 0.000
PT10: # decision nodes $\in$ Lq AND # i/o stmts $\in$ Fq,

R = 1.0, O = 11, S = 0.001

PT11: # operators/stmt ∈ Fq AND # calls ∈ Fq,
R = 1.0, O = 9, S = 0.002
PT12: # operators/stmt ∈ Fq AND # format/stmt ∈ Fq,
R = 1.0, O = 6, S = 0.016
PT13: # operators/stmt ∈ Fq AND # functions ∈ Lq,
R = 1.0, O = 8, S = 0.004

• Examples of Non-Significant Reliable Patterns

PT14: # stmts ∈ Tq AND # format/stmt ∈ Fq,
R = 1.0, O = 2, S = 0.25
PT15: # stmts ∈ Tq AND # i/o stmt/stmt ∈ Fq,
R = 1.0, O = 2, S = 0.25
PT16: # stmts ∈ Tq AND # i/o stmts ∈ Fq,
R = 1.0, O = 2, S = 0.25
PT17: # stmts ∈ Tq AND # i/o stmts ∈ Sq,
R = 1.0, O = 4, S = 0.0625
PT18: # operators/stmt ∈ Fq AND # funct/stmt ∈ Lq,
R = 1.0, O = 4, S = 0.0625

• Example of a Non-Significant Non-Reliable Pattern

PT19: # stmts ∈ Tq AND # functions ∈ Tq,
R = 0.0, O = 1, S = 1.000

High Fault Density Components

• Examples of Significant Reliable Patterns

PT1: # lines ∈ Fq AND # comment/stmt ∈ Tq,
R = 1.0, O = 11, S = 0.001
PT2: # stmts ∈ Fq AND # comment/stmt ∈ Tq,
R = 0.94, O = 17, S = 0.000
PT3: # format/stmt ∈ Lq AND # comment/stmt ∈ Tq,
R = 1.0, O = 10, S = 0.001
PT4: # decisions nodes ∈ Fq AND # call/stmt ∈ Lq,
R = 0.95, O = 21, S = 0.000
PT5: # stmts ∈ Fq AND # calls ∈ Sq,
R = 0.94, O = 18, S = 0.000
PT6: # stmts ∈ Fq AND # i/o stmt/stmt ∈ Sq,
R = 1.00, O = 13, S = 0.000
PT7: # stmts ∈ Fq AND # operand/line ∈ Sq,
R = 1.00, O = 20, S = 0.000
PT8: # stmts ∈ Fq AND # operand/stmt ∈ Sq,
R = 1.00, O = 18, S = 0.000
PT9: # stmts ∈ Fq AND # i/o variable/line ∈ Fq,
R = 1.00, O = 27, S = 0.000
PT10: # stmts ∈ Fq AND # operators ∈ Sq,
R = 0.91, O = 11, S = 0.006
PT11: # operator/stmt ∈ Lq AND # assign/stmt ∈ Lq,
R = 1.0, O = 6, S = 0.015

As shown in the above results, significant reliable patterns can be recognized and differentiated from the non-reliable / non-significant ones. Therefore, significant reliable patterns can be identified and used with confidence for both prediction and interpretation. For instance, if we take pattern PT1 for low density components, we observe a reliability of 100% based on 18 occurrences. This produces a very good pattern significance. The predictions generated by this pattern can therefore be considered very reliable and used with confidence. Both the OSR patterns and the logistic regression model yield an average classification correctness of 82%. This result is very encouraging considering that the class creation process used (i.e. dividing the range in quartiles) was primitive and that the explanatory variables available are all continuous (which is an important advantage for the logistic regression model). Moreover, note that the OSR process is entirely automated.

The patterns produced by OSR are not always easy to interpret. Interpretation of patterns (or any other stochastic model) requires expert knowledge. However, in the next subsections, we provide some rules for reading and interpreting the above patterns. Some pattern merging results are also provided.

6.2 Pattern Interpretation Rules

Interpretation of patterns is much easier than interpreting regression coefficients. First, OSR takes into account the fact that an explanatory variable can have a strong impact in a certain context (defined by the predicates in preceding positions) and not be relevant in another one. Second, if strong associations exist in a given context, then the pattern merging process makes it apparent by creating composite predicates (see examples in section 6.3). The variation of reliability generated by a particular predicate can help assess the significance of the impact of an explanatory variable (on the dependent variable) when the explanatory variable belongs to a certain class of values within a certain context. Let us take the following pattern as an example: #stmts ∈ Lq AND #calls ∈ Fq which yields a reliability of 100%. However, #stmts ∈ Lq alone only yields a reliability of 88%.

This result suggests that #calls ∈ Fq is a relevant predicate in the context where #stmts ∈ Lq because it shows a significant impact on the fault density.

However, a pattern must always be interpreted in context. In some contexts (e.g. #stmts ∈ Fq), a variable (e.g. #operators) may not take on the full range of values. The interpretation of patterns like pattern PT10 for high density components must be done carefully: #operators ∈ Sq may be interpreted as a "rather large" number of operators because in the context #stmts ∈ Fq, very few components show either #operators ∈ Tq or #operators ∈ Lq (i.e. # stmts is strongly associated with # operators). Therefore, the OSR process did not select patterns like #stmts ∈ Fq AND #operators ∈ Tq since they yielded subsets that met the termination criteria. This example shows that even though interpreting patterns is always simple, it requires the support of a tool .

6.3 Pattern Merging Results and Interpretation of Recognized Patterns

In this section, we intend to show how the merging process can help to group similar raw patterns into composite patterns and therefore provide more easily interpretable information. If we simplify the raw patterns generated by OSR using the merging criteria: $\text{Phi}^2 = 0.40$ and critical probability value of 0.0005, we get a set of composite patterns for each of the dependent variable classes. In order to illustrate the point, we first show some of the intermediate steps of the merging process. Then we give two composite patterns: CP1 and CP2 (formed by the merging process), which characterize low fault density components.

For example, low density component patterns PT1 and PT2 can be merged based on the similar classes principle. They both show the same first predicate: # stmts $\in$ Lq. Their second position predicate shows the same variable # calls and two neighboring classes (Fq and Sq). Since they do not show a statistically significant difference is distribution (critical probability value = 0.0005), then they can be merged in: #stmts $\in$ Lq AND # calls < MEDIAN.

Similarly, low density component patterns PT3 and PT4 can be merged based on the similar predicate principle. They both show the same first position predicate and their second position predicates are strongly associated ($\text{Phi}^2 = 0.57$). Therefore, they can be merged in: #stmts $\in$ Lq AND (#formats/stmt $\in$ Fq OR #I/O stmts/stmt $\in$ Fq).

This merging process is repeated until no more merging is possible according to the user's criteria. CP1 and CP2 are the final resulting composite patterns which characterize low fault density components:

CP1: SIZE_HIGH AND CALLS & I/O_LOW,
R = 99% , O = 169, S = 0.000

CP2: SIZE_HIGH AND FUNCT_HIGH,
R = 86%, O = 43, S = 0.000

where the composite predicate SIZE_HIGH is defined as:

$$\text{SIZE\_HIGH} \Leftrightarrow \begin{pmatrix} \text{\# statements} \in \text{Fq OR \# statements} \in \text{Sq} \\ \text{OR \# formats} \in \text{Lq OR \# decision nodes} \in \text{Lq} \\ \text{OR \# operators / stmt} \in \text{Fq} \end{pmatrix}$$

and, in the context where SIZE_HIGH is true, the following composite predicates are formed:

$$\text{CALLS \& I/O\_LOW} \Leftrightarrow \begin{pmatrix} \text{\# calls} \in \text{Fq OR \# calls} \in \text{Sq} \\ \text{OR I/O stmts/stmt} \in \text{Fq OR \# formats/stmt} \in \text{Fq} \\ \text{OR \# I/O stmts} \in \text{Fq OR \# I/O stmts} \in \text{Sq} \end{pmatrix}$$

$$\text{FUNCT\_HIGH} \Leftrightarrow \begin{pmatrix} \text{\# functions} \in \text{Tq OR \# functions} \in \text{Lq} \\ \text{OR functions/stmt} \in \text{Tq OR functions/stmt} \in \text{Lq} \end{pmatrix}$$

CP1 and CP2 actually define classes of raw patterns that are assessed equivalent according to the user-defined criteria. Some of the low density patterns presented in section 6.1 belong to CP1: PT1, PT2, PT3, PT4, PT8, PT9, PT10, PT11, PT12, PT14, PT15, PT16, PT17 and others to CP2: PT7, PT13, PT18, PT19. Both of the composite patterns suggest that large components are likely to have low fault densities. This agrees with a study conducted by Basili and Perricone [BP84]. This may be partially explained by the fact that low operator densities seem to be strongly associated with large components. CP1 suggests that a low number of function calls or a low number of I/O statements increase the probability of having a low fault density. CP2 indicates that a large component showing a high density of functions is likely to show a low fault density.

Merging patterns is always desirable. It allows us to combine related, rare, isolated patterns to more significant patterns and thereby group together trends which capture essentially the same phenomenon. This makes the generated composite patterns easier to interpret and gives the user a more abstract and general view of the results. Also, as we have seen, patterns with a small number of occurrences cannot be trusted (even though they show good reliabilities) because of their weak level of significance. However, if these patterns are shown to be strongly associated with other reliable patterns, then the significance of the generated composite pattern increases. This allows us to gain more trust in rare reliable patterns based on the calculated composite pattern's level of significance. However, this should be used very carefully and needs further investigation.

7. Conclusion

Based on the above experimental results, building useful models for assessing the fault density of software components, based upon early available simple metrics in the presence of noisy data appears possible. Whenever OSR generates a very reliable and significant pattern, the prediction can be used with confidence. To the contrary, if the pattern is not a reliable and significant one, an alternative modeling method such as logistic regression may give a more believable prediction. We have seen that problems such as partial information in the data set can be accommodated for by assigning a relative goodness to each prediction. Also, the patterns appear to be easier to interpret than regression coefficients and correlation matrices which are the usual outputs of regression analysis. This is due mainly to the fact that OSR produces symbolic / logical expressions where the notion of context is introduced by considering the order of the predicates. Also, the merging process helps the user look at the model at various level of abstraction. From a more general perspective, based on previous [BBT91, BP92] and current experimental results, OSR is a data analysis framework that successfully integrates statistical and machine learning approaches in empirical modeling with respect to specific software engineering needs. However, while the experimental results thus far have been encouraging, many aspects of the processes involved in OSR are still to be optimized. Such processes include, by order of importance, EV class definition, the refinement and automation of the merging process, support for pattern interpretation, the attribute selection process and the selection of termination criteria.

8 Acknowledgments

We thank Gianluigi Caldiera, Denis Oberkampf, William Thomas and especially Sandro Morasca for their excellent suggestions. We also thank the referees for their insightful comments.

References

[AG90] Alan Agresti, "Categorical Data Analysis", Wiley-interscience, 1990

[BP84] V. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, vol. 27, no. 1, January 1984.

[BR88] V. Basili and H. Rombach, "The TAME Project: Towards Improvement-oriented Software Environments", IEEE Trans. Software Engineering 14 (6).

[BS87] V. Basili and R. Selby, " Comparing the Effectiveness of Software Testing Strategies", IEEE Trans. on Software Engineering 13 (12).

[BR84] L. Breiman et al, "Classification and Regression Trees", Wadworth & Brooks, 1984.

[BBT91] L. Briand, V. Basili and W. Thomas, "A Pattern Recognition Approach to Software Engineering Data Analysis", IEEE trans. Software Eng., Special issue on software measurement principles, techniques and environments, November 1992.

[BP92] L. Briand and A. Porter, "An Alternative Modeling Approach for Predicting Error Profiles in Ada Systems", European conference on quantitative evaluation of software and systems (EUROMETRICS'92), Brussels, Belgium, April 1992.

[CA88] J. Capon, "Statistics for the Social Sciences", Wadworth publishing company, 1988.

[CE87] J. Cendrowska, "PRISM: An Algorithm for Inducing Modular Rules", Journal of Man-Machine Studies, 27, pp.349.

[DG84] W. Dillon and M. Goldstein, "Multivariate Analysis", John Wiley & sons, 1984.

[HL89] D. Hosmer and S. Lemeshow, "Applied Logistic Regression", John Wiley & sons, 1989

[M83] R. Michalski, "Theory and Methodology of Inductive Learning." In R. Michalski, J. Carbonell & T. Mitchell (Eds.), Machine learning (Vol. 1). Los Altos, CA: Morgan Kaufmann.

[M89] J. Mingers, "An Empirical Comparison of Selection Measures for Decision-tree Induction", Machine learning 3, pp.319, 1989.

[Q79] J. Quinlan, "Discovering Rules by Induction from Large Collections of Examples", In D. Michie (Ed.), Expert System in the microelectronic age. Edinburg University Press, 1979.

[Q86] J. Quinlan, "Induction of Decision Trees", Machine learning 1, Number 1, pp.81, 1986.

[SP88] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision trees for Software Resource Analysis", IEEE trans. Software Eng., 1988.

Appendix: Definition of the Generalization Algorithm (notation consistent with section 2.2)

This generalization process can be formalized using the following definitions and algorithms:

• Definition A1: We define a composite predicate (cp) as $cp = \bigcup p,\ p \in PD$, which the set of all predicates. Composite predicates can be combined to form other composite predicates. Thus, we define $cp_{i \cup j} = cp_i \cup cp_j$.

• Definition A2: An *association coefficient* a_{ij}^{PSS} is an assigned statistical degree of association between cp_i and cp_j where PSS is the data set used to determine this association. Let us assume the two following data subsets:

$$PSS_i = \{pv \in PSS | cp_i \text{ is true}\}$$
$$PSS_j = \{pv \in PSS | cp_j \text{ is true}\}$$

A two row-two column contingency table is defined, where the subsets characterizing each row and column are respectively PSS_i, PVS - PSS_i, PSS_j, PVS - PSS_j. Based on this table, a Chi-Square based statistic (i.e. Pearson's Phi) defining the degree of association between the two subsets is calculated and assigned to a_{ij}^{PSS}.

• Definition A3: A *context* is a conjunction of a set of composite predicates that defines $PSS \subseteq PVS$. This defines the data subset on which an association coefficient is calculated and therefore its domain of validity.

• Definition A4: An *association matrix* A_{mn}^{C} is a square matrix of association coefficients calculated in a context C, where the rows / columns represent all possible predicates.

<u>example</u>: $A_{nm}^{cp_k \wedge cp_l}$ contains all a_{ij}^{PSS}
where $\forall\ pv \in PSS,\ cp_k \wedge cp_l$ is true.

• Definition A5: Two composite predicates cp_i and cp_j are said to be *associated* in the context of C if $a_{ij}^{PSS} \geq$ some minimal level of association.This will be denoted as $cp_i \approx cp_j$.

• Definition A6: A *predicate tree* is a tree representation of the patterns generated during the Development Environment Analysis (i.e. DEA) process. As mentioned is Section 3.2, DEA produces a set of patterns representing the observed trends in the historical data set. It is expected that a significant number of these patterns will be duplicated or similar. This representation is a compact way of representing the specific pattern set (SPS). Each path of a predicate tree represent a pattern generated by DEA. (see Figure 3)

PATTERN SET

$EV_1 \in Class_1$ AND $EV_3 \in Class_2$
$EV_4 \in Class_1$ AND $EV_2 \in Class_3$
$EV_1 \in Class_1$ AND $EV_3 \in Class_2$
$EV_4 \in Class_1$ AND $EV_5 \in Class_2$

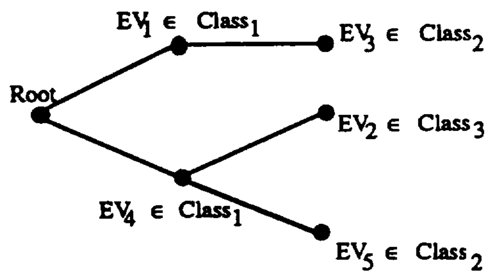

Figure 3: Example Predicate Tree

Notice that the root of the predicate tree is a "dummy" predicate which can be thought of as the *identity predicate* cp_I (i.e. $cp_i \wedge cp_I \Leftrightarrow cp_i$). Note that in the above example, all of the predicates are singleton. This represents a predicate tree before any generalization. Branches will be merged and composite predicates created at the nodes during the generalization process.

• Definition A7: The *maximum merging depth* (user defined) is the depth in the predicate tree to which generalization is to be performed. It defines the observation depth of the patterns by the user.

• Definition A8: Two composite predicates cp_i, cp_j are said to be "mergeable neighboring composite predicates" if the following conditions are fulfilled:

(1) There exist two predicates p_x: $Xi \in class_{ik}$, p_y: $Xi \in Class_{it}$ such that p_x and p_y are one of the disjunctive predicates of cp_i and cp_j, respectively.

(2) $Class_{ik}$ and $Class_{it}$ are neighboring classes on variable X_i range.

(3) cp_i and cp_j yield the same classification, show a difference of reliability below DR and a maximum

C-2

pattern level of significance S (i.e. DR and S are fixed by the user).

If these three conditions are true, then mncp(cp_i, cp_j, S, , DR) is true.

In order to define the generalization algorithm based on the above definitions, we assume that it starts with the procedure call: Generalize(predicate tree, root, cp_I, 0, PHI, DR)

We can now define the Generalize algorithm as follows:

```
procedure Generalize (predicate tree, node, context, current depth, PHI, DF)
    (1) If the node is a terminal node of the predicate tree
    OR if depth > maximum merging depth then
      RETURN
    (2) while
      $\exists\, cp_i, cp_j$ such that mncp ($cp_i$, $cp_j$, S, DF) do
      merge(predicate tree, node, $cp_i$, $cp_j$)
    (3) calculate $A^{context}_{mxm}$, the association matrix with all
    $cp_i$'s, $i \in \{1,\dots,m\}$, in context.
    (4) while $\exists\, cp_i, cp_j$ such that $cp_i \approx cp_j$ do
      . select $cp_i$ and $cp_j$ such as $a^{context}_{i,j}$ is the strongest
      association in $A^{context}_{mxm}$
      . merge(predicate tree, node, $cp_i$, $cp_j$)
      . recalculate $A^{context}_{m-1\,x\,m-1}$, the association matrix for
      $cp_i, \dots, cp_{i-1}, cp_{i+1}, \dots, cp_{j-1}, cp_{j+1}, \dots, cp_m$,
      $cp_{iUj}$ in context.
    (5) for each successor of node in predicate tree
    Generalize (predicate tree, successor, context ^ $cp_{node}$,
                                          depth+1, PHI, DF)
end Generalize
```

In step (4), a call is made to procedure merge defined merge as follows:

```
procedure merge (predicate tree, node, $cp_i$, $cp_j$)
    $cp_i$ and $cp_j$ are successors of node
    (1) Combine $cp_i$ and $cp_j$ to form a single node
    $cp_{iUj}$
    (2) Combine all like subpaths rooted at $cp_{iUj}$
    end merge
```