T-SE/19/11//13112D

Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components

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

Reprinted from
IEEE TRANSACTIONS ON SOFTWARE ENGINEERING
Vol. 19, No. 11, November 1993

Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components

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

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 frequency 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 the Optimized Set Reduction approach for constructing such models, intended to fulfill specific software-engineering needs. Our approach to classification is to measure the software system and build multivariate stochastic models for predicting high-risk system components. We present experimental results obtained by classifying Ada components into two classes: is or is not likely to generate faults during system and acceptance test. Also, we evaluate the accuracy of the model and the insights it provides into the error-making process.**

Index Terms— **Classification tree, data analysis, fault-prone Ada components, logistic regression, machine learning, Optimized Set Reduction, stochastic modeling.**

I. Introduction

IT has been noted that a small number of software components are responsible for a disproportionately large number of faults in any large-scale system [5], [23], [28]. Therefore, if we can identify components likely to produce a large number of faults, we can concentrate the verification and testing processes on them. This allows us to optimize the reliability of our software system with minimum cost. To do this, we build quantitative models that predict which components are likely to contain the highest concentration of faults. However, building such models is a difficult task: it is often the case in software engineering that the data which is collected are minimal, incomplete, and heterogeneous [12]. This presents several problems for model construction and interpretation (e.g., small data sets, inaccurate models, outliers). Therefore, we need a modeling process that is robust to these problems, allows for the reliable classification of high-risk components (those that have a high probability of generating a fault during system or acceptance test), and aids in the understanding of the causes of this high risk. This understanding is important because it can give us insight into the software-development process, allowing us to take remedial actions and make better process decisions in the future.

Manuscript received January 5, 1993; revised July 23, 1993. This work was supported in part by NASA Grant NSG 5123 and NSF Grant 01-5-24845. Recommended by F. Bastani.

The authors are with the Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, College Park, MD 20742.

IEEE Log Numer 9213112.

In this context, we will examine the use of the following modeling approaches:

- Logistic regression, which is one of the most commonly used classification techniques [1], [21]. This technique has been applied to software engineering modeling [23], as well as other experimental fields, and will therefore be used as a baseline for comparison in this paper. Many assumptions and constraints inherent to this technique make it difficult to apply in a software engineering context: 1) nonmonotonicity of the probability density function on the explanatory variable range and 2) interactions between explanatory variables are difficult to take into account when performing exploratory data analysis with numerous explanatory variables.
- Classification trees, described in [8]. They are used to address software-engineering modeling issues in [24], [28]. A review may be found in [12], [15]. Their strength stems from their simplicity and readability. Their weaknesses come from a lack of ability to extract and use all statistically significant trends and a tendency to include nonrelevant and nonsignificant information in the tree.
- Optimized Set Reduction (OSR), which has been developed at the University of Maryland [12] in the framework of the TAME project [7] and has already been applied to several software engineering applications [10]–[12]. It is partially based on both machine learning principles [8], [25] and univariate statistics [19]. Our motivation for developing OSR, and a tool to support it, was to design a data analysis approach that matches, to the extent possible, the specific needs of multivariate empirical modeling for software engineering [12]. OSR generates logical expressions that represent patterns in a data set. For instance, consider the following example of a simple pattern (characterized by a logical expression) related to high-fault concentration:

Example 1: A compilation unit that imports numerous declarations from outside the subsystem in which it is developed, that shows a large average statement nesting level and an intense use of global variables, is likely to generate fault reports during system and acceptance testing. The corresponding logical expression characterizing this class of compilation units

0098–5589/93$03.00 © 1993 IEEE

would be:

$$\text{NONLOC_IMP} = \text{High} \wedge (\text{NESTING} = \text{High} \wedge \text{GLOBALS} = \text{High})$$

In this paper, we intend to show that OSR may be used as an alternative to logistic regression or classification trees to generate empirical models of risk within a software system and that it can yield more accurate results. We will discuss issues related to the interpretation of the generated models. In particular, we will demonstrate how OSR can be useful in 1) identifying characteristics of high-risk components in a large Ada system and 2) providing some understanding about how faults originate during the software-development process

In Section II, we present an evolved version of the OSR algorithm (an earlier version of the OSR approach was applied to project cost estimation and published in [12]), which is intended to make OSR models more accurate and easier to interpret. Specifically, the new algorithm improves the interpretability and the accuracy of the models in three ways. First, it provides a mechanism for dealing with the discretization of the explanatory variable ranges in an automated way. This better supports the requirement that our models need to be able to handle the problem of heteroscedascity (see R5 in [12]). Secondly, we provide OSR with the ability to work with *conjunctive predicates* (which will be called *predicates* in this paper), allowing our models to elicit the effects of combinations of variables not visible in the previous version of OSR. Finally, we provide support for recognizing similarities among patterns, which aids the user in model interpretation. These second and third adaptations help OSR deal with the requirement that our models are able to handle interdependencies and interactions among the explanatory variables (see R4 in [12]).

Also, in contrast to [12], this paper applies the OSR modeling technique to the issue of classifying Ada components as either low or high risk, as opposed to project-cost estimation (prediction on a continuous range). Accordingly, we use logistic regression and classification trees as a baseline for evaluating the OSR results. (Preliminary and partial results of this research were presented in [10] based on the analysis of FORTRAN systems.)

In Section III, we present a validation of the OSR process, which is based on constructing models using data from a large Ada system developed at the NASA Goddard Space Flight Center. In Section III.B we compare the generated OSR models to both logistic regression and classification-tree models with respect to their accuracy. In Section III.C, we discuss the interpretability of the OSR models. Finally, in Section IV, we outline the main conclusions of this paper and define the future directions of the research.

II. Optimized Set Reduction

Assume we want to assess a characteristic of an object. We will refer to this characteristic as the dependent variable (Y). The object is represented by a set of explanatory (known or assessable) variables (called X's). These variables can be either discrete or continuous. Also, assume we have a historical data set containing a set of experiences that contain the previously cited X's plus an associated actual Y value. Our goal will be to determine which subset of experiences from the historical data set provides the best characterizations of the current object to be assessed.

Example 2: Assess the expected frequency of faults (Y) that will be detected during system and acceptance test within a particular compilation unit. For instance, the X's may be: complexity metrics, system architecture metrics, or developer-related evaluation of skills.

A. The OSR Process

First, we will introduce new terminology in an attempt to both formalize the intuitive concepts related to empirical modeling and give those concepts a firm grounding in the OSR context. Subsection II.A.1 presents the notions informally to provide the reader with some intuition about the method. The rest of Section II will be more structured and formal in order to define more complex notions without ambiguity. Whenever needed, definitions will be formal specifications, whereas others will be in algorithmic form.

1) Basic Definitions: Assume we have a historical data set consisting of n experiences, where each experience consists of a value for a single dependent variable (Y) and a set of values corresponding to a set of m explanatory variables (EV $= \{X_1, X_2, \cdots, X_m\}$). We define the term *pattern vector* to mean one of these such experiences. Assume the dependent variable's value domain ($\text{dom}(Y)$) is divided into a set of equivalence classes that can be either intervals (if the Y is continuous) or categories (if the Y is discrete). Each explanatory variable has its own value domain ($\text{dom}(X_i)$), which, like $\text{dom}(Y)$, is divided into a set C of equivalence value classes C $= \{\text{Class}_{i1}, \text{Class}_{i2} \cdots \text{Class}_{ik}\}$. We define a *measurement vector* to be a pattern vector without the dependent variable Y. (Note that a measurement vector can be used to represent an object whose dependent variable value is not known but is of interest and which we wish to assess.) The measurement vector value domain is

$$\text{MV} = \times_{i\in\{1\cdots m\}} \text{dom}(X_i).$$

Likewise, the pattern-vector value domain (i.e,. the domain of the vectors in the data set) can be represented as PV $= \text{dom}(Y) \times \text{MV}$. We define PVS $\subseteq$ PV to be a *pattern vector set*, representing the *historical data set*.

Example 3: Suppose (Size = 100 LOC's, Function_type = computation) is a measurement vector characterizing a compilation unit. Assuming Y is #faults, (#faults = 6, Size = 100 LOC's, Function_type = computation) is a pattern vector characterizing a particular testing experience on a compilation unit.

At the very heart of the OSR process is what we call a *singleton predicate*. We define a singleton predicate to be a pair with the following form: (X_i, Class_{ij}), meaning that explanatory variable X_i has a value belonging to $\text{Class}_{ij} \subseteq \text{dom}(X_i)$. A singleton predicate (also written $X_i \in \text{Class}_{ij}$) is said to be TRUE for a measurement vector if that vector's explanatory

variable X_i value is an element of Class_{ij}, otherwise the singleton predicate is said to be FALSE for that vector.

Example 4: Size $\in$ [50, 200) is a singleton predicate.

Now that we have defined the notion of a singleton predicate, we can define other elements of OSR built on this notion. For instance, we can define a *conjunctive predicate* (denoted *Pred* and simply called a *predicate* from here on) as the conjunction of singleton predicates. We will consider a predicate to be a set of singleton predicates where the conjunction is implicit. A predicate is said to be TRUE for a given measurement vector if each of its constituent singleton predicates is TRUE for that vector. (Note that by defining a predicate to be a set (conjunction) of singleton predicates gives OSR the ability to elicit some of the complex interdependencies that exist between the explanatory variables, see requirement R4 in [12].)

Example 5: Size $\in$ [50, 200) $\wedge$ Function_type $\in$ {computation} is a predicate.

A predicate may be used to characterize sets of pattern vectors. For example, if we define IS_TRUE(Pred, pv) to yield TRUE if Pred is a true logical expression for the pattern vector pv, (i.e., each singleton predicate in Pred is true for pv), then we can define a predicate Pred and a subset PSS of the historical data set (PVS) such that IS_TRUE(Pred, pv) yields TRUE for each pv in *PSS*. Similarly, we define SUBSET(PSS, Pred) to denote the subset of PSS characterized by Pred. Also, we define PSS to be equivalent to SUBSET(PSS, TRUE). Finally, MEMBER(X, Pred) yields the value TRUE if the variable X appears anywhere in Pred, FALSE otherwise.

2) Optimal Subsets of Experiences: In this section, we rigorously define the notion of "optimal subset of experiences" by defining the function OPT that extracts these subsets from a given historical data set. We will see in the next section that OPT is not directly implementable. Nonetheless, this definition should help the reader understand our goals at a first glance. These definitions, by their very nature, are somewhat terse. However, the accompanying explanations should help the reader get an intuitive understanding of the process.

Definition 1: Normalized Entropy $H(\text{PSS}, Y)$. This is the information-theory definition of entropy that characterizes distributions, normalized to yield a value between 0 and 1. This concept is commonly used in machine learning [22] to assess the level of information provided by a distribution on a continuous or discrete range. It yields a value 0 when unambiguous information is provided and 1 when no information is provided.

$$H(\text{PSS}, Y) = -\sum_{\text{Class } Y_j \in C} p(\text{PSS}, \text{Class } Yj) \log_{|C|} p(\text{PSS}, \text{Class } Yj)$$

where

- PSS is a set of pattern vectors
- Class Y_j is a class defined on dom(Y)
- $p(\text{PSS}, \text{class } Y_j)$ is the prior probability that a vector that is an element of PSS has a dependent variable value belonging to the dependent variable class Y_j.
- C is the set of classes defined on the range of Y.

Definition 2: DIFFDIST(PSS_i, PSS_j, Y). This function yields TRUE if the two sets of pattern vectors characterized by PSS_i and PSS_j show a statistically significant DIFFerence in DISTribution on the dependent variable (Y) range and is FALSE otherwise. This function is based on binomial tests for proportions and is better described in [12]. The statistical level of significance used as a threshold between TRUE and FALSE is subjective and is therefore defined by the user (e.g., 0.05, 0.1). ♦

The next defined function determines whether or not a pattern vector subset PSS is of any interest in order to predict a measurement vector mv

Definition 3: VALID(PSS, mv). This function yields TRUE if at least one predicate is TRUE for all the pattern vectors in PSS and for the measurement vector mv.

$$\begin{aligned} &\text{PSS} \subseteq \text{PVS} \wedge mv \in \text{MV} \wedge \exists \text{Pred such that} \\ &(\forall pv \in \text{PSS}, \text{IS_TRUE}(\text{Pred}, pv) \wedge \text{IS_TRUE}(\text{Pred}, mv)) \\ &\Rightarrow \text{VALID}(\text{PSS}, mv) \end{aligned}$$

Based on Definitions 1 and 2, the next function (EMIN) determines if a subset $\text{PSS}_j \subset \text{PSS}$ is the subset of PSS with maximum predictive power among all possible subsets, i.e., the one with minimum entropy and showing a statistically significant difference in distribution as compared to PSS.

Definition 4: EMIN(PSS, PSS_j, Y). This function yields TRUE if PSS_j, a subset of PSS, shows a significantly different distribution from PSS on the Y range (based on a predefined level of significance and according the result of the function DIFFDIST) and for all other subsets PSS_k of PSS showing a statistically significant Y distribution, $H(\text{PSS}_j, Y) \leq H(\text{PSS}_k, Y)$. EMIN stands for: Entropy is MINimum. In other words, EMIN tells us if PSS_j characterizes a subset with minimal possible entropy and that this low entropy is not likely to be due to chance.

$$\begin{aligned} &\text{PSS} \subset \text{PVS} \wedge \text{PSS}_j \subset \text{PSS} \\ &\quad \wedge (\text{DIFFDIST}(\text{PSS}_j, \text{PSS}, Y) \wedge (\forall \text{PSS}_k \\ &\quad \subset \text{PSS}, k \neq j, \text{DIFFDIST}(\text{PSS}_k, \text{PSS}, Y) \\ &\quad \wedge H(\text{PSS}_j, Y) \leq H(\text{PSS}_k, Y))) \\ &\quad \Rightarrow \text{EMIN}(\text{PSS}, \text{PSS}_j, Y) \end{aligned}$$

♦

Based on the above definitions, the function OPT extracts the optimal subsets of experiences, i.e., the ones showing a maximum predicitive power for mv on the Y range.

Definition 5: OPT(PVS, mv, Y). OPT yields a set of OPTimal subsets of pattern vectors of PVS (the historical data set) based on the definitions presented above. These subsets are characterized by predicates that are built based upon known information (i.e., mv) and show a minimal entropy. They can therefore be used for predicting the value of Y with respect to mv.

Example 6: In Fig. 1, based upon a given measurement vector (mv) and a given historical data set, the optimal subset extracted by OPT and characterized by the predicate on the left-hand side of Fig. 1 indicates a strong probability for Y to lie in the interval $Y2$. This may be used for predicting the class

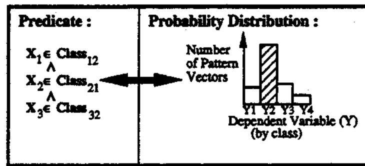


Fig. 1. Classification with extracted subsets.

where the object described by mv is likely to lie. Also, if Y is defined on a continuous scale, the optimal subset expected value may be used as a prediction.

Based on the primitives defined above, OPT may be defined as follows:

$$\text{OPT}(\text{PVS}, mv, Y) = \{\text{PSS} \mid \text{PSS} \subseteq \text{PVS} \wedge \text{VALID}(\text{PSS}, mv) \wedge \text{EMIN}(\text{PSS}, \text{PVS}, Y)\}.$$

The function OPT as defined above defines optimal subsets of experiences with minimal entropies and characterized by optimal predicates. However, this is just a first step in the definition of an optimal search algorithm to extract datasets' patterns since there are several reasons why this simple function is not fully adequate to build empirical models to fulfill our needs. Some of these reasons are simply computational in nature while others are related to the loss of useful information.

1) The number of possible singleton-predicate combinations makes the execution time of the search of optimal predicates prohibitive without a search strategy.
2) We are not only interested in the optimal subsets extracted by OPT but also by the predicates that characterize them. We want each generated predicate to contain only singleton predicates that have a *significant* impact on the resulting distribution entropy (see Fig. 1). Thus, we can minimize the information necessary to identify optimal subsets and make the predicates more interpretable.
3) We need to extract information about the relative impact of the various singleton predicates within the optimal predicates.
4) The conditions under which singleton predicates or predicates appear relevant have to be determined.

Therefore, we will now define an algorithm that addresses these issues, discussing its relationship to the function OPT. This is the Optimized Set Reduction process that can roughly be described by a three-step recursive algorithm where entropy is optimized in a stepwise manner.

B. The OSR Algorithm

The goal of the OSR algorithm is to produce a set of *patterns* that characterize the trends observable in the historical data set while addressing the four modeling issues mentioned above. In this context, the notion of pattern is based on the notion of predicate as defined above while addressing some of the mentioned modeling needs. This definition of pattern intends to be both useful for predicting and suitable to interpretation.

In subsection II.B.2, we shall describe the OSR algorithm in detail. However, before doing so, we need to define a number of preliminary concepts used in the algorithm.

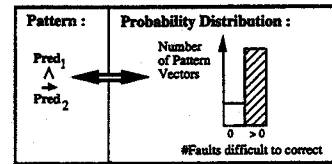


Fig. 2. Classification using patterns.

1. Preliminary Definitions: Definition 6: OSR Pattern. As mentioned above, OSR generates patterns. A pattern is an *ordered* conjunction of predicates that characterizes a subset of PVS that shows a minimal entropy distribution. The notion of ordering will be represented by the "ordered AND" symbol $\overset{\wedge}{\rightarrow}$. It is logically equivalent to the symbol $\wedge$ with the exception that predicates to the right of a $\overset{\wedge}{\rightarrow}$ symbol are relevant only when all predicates to the left of the symbol are already TRUE. The notion of order is introduced here to capture information about the conditions under which a predicate is relevant. We will call the ordered expression to the left of a given predicate in a pattern the *context* of the predicate. This addresses issue R4 mentioned in the Introduction.

Example 7: Definition of two predicates

$$\text{Pred}_1 = \text{SUBSYSTEM} \in \text{REAL-TIME CONTROL} \wedge \text{SUBSYSTEM} \in \text{LARGE}$$
$$\text{Pred}_2 = \#\text{GLOBAL VARIABLES} \in \text{LARGE}.$$

If we assume the pattern $\text{Pred}_1 \overset{\wedge}{\rightarrow} \text{Pred}_2$ was generated by OSR, we can see that this pattern characterizes a pattern vector set suggesting a high *risk* that is defined, in this particular example, as the probability of containing errors difficult to correct during the test phases see (Fig. 2).

This pattern $\left(\text{Pred}_1 \overset{\wedge}{\rightarrow} \text{Pred}_2\right)$ has specific interpretation associated with it. Pred_1 is a non-singleton predicate and Pred_2 is relevant within the context of Pred_1. This pattern implies the following interpretation. If a subsystem is both large and real time, then it is significantly more likely to be of high risk than a random subsystem. However, it does NOT suggest that either real-time subsystems or large subsystems independently increase the probability that a subsystem will be of high risk. Also, within the context of large, real-time subsystems, subsystems with a large number of global variables have a significantly greater probability of being high risk than those with a small number of global variables. However, this pattern does NOT suggest that a large number of global variables has a significant impact on the probability that a subsystem will be of high risk outside the context of large, real-time subsystems. (More details concerning pattern generation and interpretation will be presented later in the paper.)

Definition 7: DISCRETIZE(PSS, X_i). Given a particular subset of pattern vectors (PSS), we want to divide/cluster the ranges/categories of the explanatory variables into a set of equivalence classes ($\text{Class}_{i1} \cdots \text{Class}_{ik}$ for the explanatory variable X_i) based on a meaningful class-creation techniques. This is used to both define singleton predicates and to better satisfy the problem of heteroscedascity, i.e., requirement R5 of [12], which states that an explanatory variable may be a

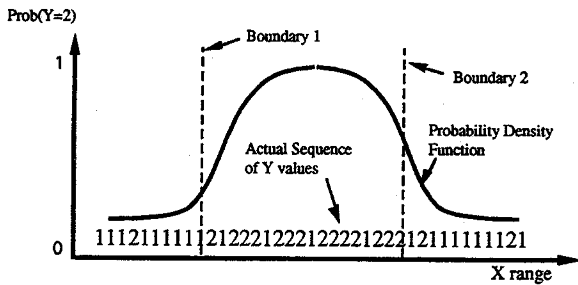


Fig. 3. Discretization process.

good predictor on a part of its range/value domain while a mediocre predictor otherwise. Clustering of discrete categories can only be performed by the user by defining taxonomies. Numerous techniques are available in the literature to create intervals on continuous/ordinal ranges (e.g., cluster analysis) [16]. However, none appears to have satisfactory properties for our problem. Therefore, classes are created for continuous/ordinal explanatory variables according to the procedure DISCRETIZE briefly presented below and described in Appendix B.

DISCRETIZE(PSS, X_i) defines classes on the range of X_i (a particular continuous or ordinal explanatory variable) based on a pattern vector subset PSS. This algorithm has the following properties:

- Either all or some of the classes should show distributions on the Y range significantly different than the distribution resulting from the union of those classes. If not, differentiating these classes and creating new pattern-vector subsets is meaningless.
- The algorithm handles monotonic and nonmonotonic underlying distributions on the Y range.
- The algorithm is not oversensitive to the addition or deletion of few pattern vectors so stable patterns are generated.

Our goal is to take into account the above constraints and to minimize the average entropy across the created classes in order to have classes as homogeneous as possible with respect to the dependent variable values of their pattern vectors. Fig. 3 illustrates the output of the algorithm. We assume an actual underlying and unknown nonmonotonic probability density function and an observed sequence of Y values on the explanatory variable X range. We also assume that two classes (1, 2) are defined on the Y value domain. Using the DISCRETIZE algorithm produces Boundary1 and Boundary2 in Fig. 3, which creates the corresponding set of three explanatory variable value classes across the X range.

Definition 8: GENERATE_SINGLETONS (PSS, mv). Let PSS represent the considered pattern vector set and let mv be a measurement vector. The classes defined by DISCRETIZE for each explanatory variable X_i give us a set of singleton predicates: $\{X_i \in \text{Class}_{i1}, \cdots, X_i \in \text{Class}_{ik}\}$. GENERATE_SINGLETONS(PSS, mv) generates the set of all singleton predicates SP such that SP $= \{\text{Pred}_i \mid \text{IS_TRUE}(\text{Pred}_i, mv)\}$.

Definition 9: SIG_PREDICATE(PSS, Pred, Y). The predicate Pred is said to be *significant* for the data set PSS if SUBSET(PSS, Pred) shows an entropy lower than the one of PSS and if their distributions on the Y range show statistically significant differences.

$$\begin{aligned}&\text{PSS} \subseteq \text{PVS} \wedge (H(\text{SUBSET}(\text{PSS, Pred}), Y) < H(\text{PSS}, Y)\\&\quad \wedge\, \text{DIFFDIST}(\text{PSS}, \text{SUBSET}(\text{PSS, Pred}), Y))\\&\quad \Rightarrow \text{SIG_PREDICATE}(\text{PSS, Pred}, Y)\end{aligned}$$

Example 8: Assuming two dependent-variable classes ([low, high]), suppose *Pred* characterizes a subset whose distribution across the two classes is [10, 7]. This subset shows an entropy lower than the entropy of PSS, which had a distribution [100, 75], but the difference is not statistically significant since the proportion of pattern vectors in each class is practically the same. Statistical tests for two population proportions [14] can be used to assess the significance of the observed difference in entropy.

Definition 10: MINIMAL(PSS, $\text{Pred}_i r$, Y). The predicate pred_i is said to be *minimal* for the pattern-vector set PSS if it characterizes a subset of PSS that shows a significantly different distribution across the Y classes and there exists no other predicate $\text{Pred}_j \Rightarrow \text{Pred}_i$ such that Pred_j characterizes a subset of PSS that shows a significantly different distribution across the Y classes. Otherwise, Pred_i contains more singleton predicates than is necessary to significantly improve the entropy and is not considered to be *minimal.*

$$\begin{aligned}&\text{SIG_PREDICATE}(\text{PSS}, \text{Pred}_i, Y)\\&\wedge\\&(\forall j, \text{Pred}_j \Rightarrow \text{Pred}_i, j \neq i,\\&\ (\neg\text{SIG_PREDICATE}(\text{PSS}, \text{Pred}_j, Y))\\&\ \Rightarrow \text{MINIMAL}(\text{PSS}, \text{Pred}_i, Y)\end{aligned}$$

Example 9: Assume that the predicate Pred_1 = SUBSYSTEM $\in$ REAL-TIME CONTROL $\wedge$ SUBSYSTEM $\in$ LARGE yields, in a defined context, an entropy of 0.5 (assumed to yield a significantly different distribution from the parent set). If Pred_2 = SUBSYSTEM $\in$ REAL-TIME CONTROL by itself yields an entropy of 0.5, Pred_1 is not *minimal.*

Definition 11: VALID_PREDICATES(PSS, PRED_c, SP, Y). Let PSS represent a set of pattern vectors and PRED_c be the set of predicates that define the context characterizing PSS. Let SP be a set of singleton predicates and Y be the dependent variable.

Assuming that the set SP has been created by using GENERATE_SINGLETONS, we generate the set of all predicates that are conjuncts of the singletons in SP and that are *minimal* with respect to PSS (as defined above), as long as they do not use any explanatory variable X that appears in PRED_c. These predicates are called *valid* and are the ones that appear potentially useful for extracting subsets of PSS with high predictive power for mv on the Y range. With respect to the implementation of this procedure, the user may restrict the search space by fixing a maximum number of singleton predicates per predicate. However, some complex

but meaningful predicates may not be extracted by doing so.

$$\begin{aligned}&\text{VALID_PREDICATES}(\text{PSS}, \text{PRED}_c, \text{SP}, Y)\\&= \{\text{Pred}_i \mid \text{MINIMAL}(\text{PSS}, \text{Pred}_i, Y) \wedge \text{Pred}_i \subseteq \text{SP}\\&\quad \wedge (\forall j, \text{Pred}_i \in \text{PRED}_c, \forall X \text{ such that}\\&\quad X \in \{X_k \mid X_k \in \text{EV} \wedge \text{MEMBER}(X_k, \text{Pred}_j)\},\\&\qquad \neg\text{MEMBER}(X, \text{Pred}_i)\}\end{aligned}$$

Definition 12: EXTRACT_SUBSETS(PSS, PRED). Let PRED be a set of predicates. A set of subsets, where each subset is characterized by one and only one predicate in the set PRED, is extracted from PSS.

$$\begin{aligned}&\text{EXTRACT_SUBSETS}(\text{PSS}, \text{PRED})\\&= \{\text{PSS}_i \mid \text{Pred}_i \in \text{PRED} \wedge \text{PSS}_i = \text{SUBSET}(\text{PSS}, \text{Pred}_i)\}\end{aligned}$$

2) The OSR Algorithm: When the dependent variable's value domain is defined on a continuous/ordinal scale, its range is assumed to be divided into intervals/classes. These classes are fixed and will be used throughout the algorithm. These intervals are usually defined according to two main criteria: the size of the data set and the specific use of the model. The larger the data set, the narrower the classes may be, so that the model can produce a more accurate response. Also, the definition of these classes must also take into account the future use of the model, e.g., they represent clusters on the Y range or a finite number of situations suggesting alternative actions.

Example 10: Assume that the range of the dependent variable (Y) is the set of all natural numbers, indicating the number of fault reports that were generated for a component during system and acceptance test. Then, we may decide to define the following dependent-variable classes:

$$\text{Class}\, Y1 = Y \text{ in } [0, 1) \qquad \text{Low-Risk Components}$$

$$\text{Class}\, Y2 = Y \text{ in } [1, +\infty) \qquad \text{High-Risk Components}$$

Let PSS be a set of pattern vectors, let mv be a measurement vector characterizing the object to be classified on the Y range, and let PRED_c be the set of predicates composing the pattern characterizing the set PSS. Recall that we cannot use OPT directly. However, OSR(PSS, mv, PRED_c, Y) heuristically returns a set of "optimal" subsets using the algorithm defined below.

OSR (PSS, mv, PRED_c, Y)

- *Step 1:* SP = GENERATE_SINGLETONS(PSS, mv)
  -- Generate a set of optimal singleton predicates based on the pattern vector set PSS and for the measurement vector mv
- *Step 2:* PRED = VALID_PREDICATES(PSS, PRED_c, SP, Y)
  -- Generate all the valid predicates based on the available set of singleton predicates SP, the current context defined by PRED_C, and its corresponding pattern vector set PSS.
- *Step 3:*

```
if PRED = ∅ -- no Predicates have
 been created at Step2
        return PSS;
 else
   {
     -- A subset is extracted for each
        valid predicate created at step 2
     -- OSR is called recursively for each
        of these extracted subsets
     for all PSS_i ∈ EXTRACT_SUBSETS
      (PSS, PRED) do
        {
        -- the context of PSS_i is now the
           context of PSS union Pred_i
        PRED_i = PRED_c ∪ Pred_i;
        -- call OSR for the subset PSS_i
        OSR(PSS_i, mv, PRED_i, Y);
        }
   }
```

Initially, call OSR (PVS, mv, $\varnothing$, Y) where PVS is the historical data set.

The OSR algorithm can be viewed as a recursive function of OPT as described below. PVS is the historical data set and mv the vector describing the object to be assessed. Let us assume we modify the definition of the function VALID, which is used to build OPT, so that the function MINIMAL is included in it. Then, VALID becomes the following:

$$\begin{aligned}&\text{PSS} \subset \text{PVS} \wedge mv \in MV \wedge \exists \text{Pred}_i \text{ such that}\\&\quad (\forall pv \in \text{PSS}, \text{IS_TRUE}(\text{Pred}_i, pv)\\&\quad \wedge \text{IS_TRUE}(\text{Pred}_i, mv)\\&\quad \wedge MINIMAL(PSS, \text{Pred}_y, Y))\\&\Rightarrow \text{VALID}(\text{PSS}, mv, Y)\end{aligned}$$

Then, assuming the definition of OPT uses this new definition of VALID, we can then define OSR in the way shown at the bottom of this page.

Note that at each level of recursion, a minimal subset of pattern vectors is extracted. These recursively nested, extracted subsets are each characterized by a predicate in a context.

$$\text{OSR}(\text{PSS}, mv, \text{PRED}_C, Y) = \left\{\begin{array}{ll}\bigcup_{\text{PSS}_i \in \text{Opt}(\text{PSS}, \text{mv}, Y)} (\text{OSR}(\text{PSS}_i, mv, \text{PRED}_C \cup \text{Pred}_i, Y)), & \text{if OPT}(\text{PSS}, \mathbf{mv}, Y) \neq \varnothing\\ \{\text{PSS}\}, & \text{otherwise.}\end{array}\right\}$$

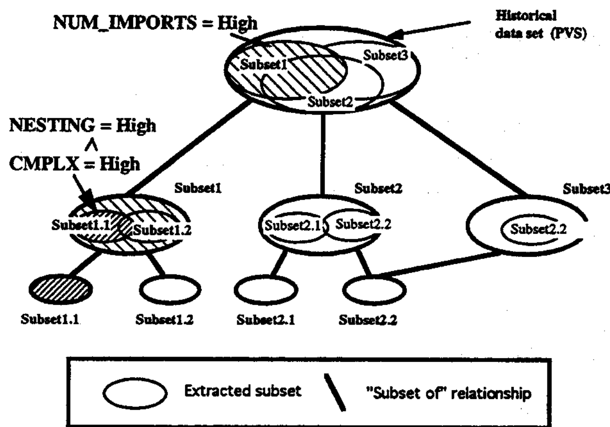


Fig. 4. An example of OSR hierarchy.

Thus, this process can be represented as a hierarchy where the ordered conjunctions of predicates are the hierarchy's branches (see Definition 6 and Fig. 4).

The subsets of PVS extracted by OSR for a particular mv may be used for the classification of Y for mv. Also, if patterns are extracted for each mv in PVS, the resulting set of patterns may be used for the interpretation of the impact of the explanatory variables on the dependent variable in a particular development environment. These issues will be addressed in the next sections.

Example 11: In Fig. 4, we can see how OSR patterns are generated during the subset extraction process. At the first (highest) level in the hierarchy, suppose ⟨NUM_IMPORTS = HIGH⟩ is a predicate that is *minimal*, causing the extraction of Subset 1. At the second level, suppose the two place predicate ⟨NESTING = HIGH ∧ CMPLX = HIGH⟩ was found to be *minimal*. Then, by tracing the hierarchy down this particular path, OSR generates the following pattern, which corresponds to the extracted subset 1.1:

$$\text{NUM_IMPORTS} = \text{HIGH} \xrightarrow{\wedge} (\text{NESTING} = \text{HIGH} \wedge \text{CMPLX} = \text{HIGH})$$

Also, each path in the hierarchy from the top set (PVS) to a bottom-level subset is marked by its own pattern. Thus, OSR creates a set of patterns, (i.e., all the paths in the hierarchy).

Each path of the hierarchy represents a path that the extraction process may have taken during OSR. Accordingly, each path is characterized by an ordered conjunction of predicates, i.e,. a pattern. Each final extracted subset (i.e., leaves of the hierarchy) forms a probability distribution across the dependent variable range This distribution is a valuable piece of information and can be used in several ways. For instance, 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's Y value to lie in. Alternatively, we may consider using a *Bayesian approach*. That is, we could define a *loss/risk function* [12] and select the dependent variable class yielding the minimum *expected loss*. Finally, note that several leaves may have distributions that yield contradictory or dissimilar trends. Therefore, several pattern classifications (i.e,. hierarchy leaves) are used to make a final global classification 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 (e.g., hierarchy branches). This is the topic of the next subsection.

C. Assessing the Accuracy of Patterns

In order to generate patterns and assess their accuracy, we use OSR in the context of the technique called *V-fold Cross Validation* [8]. For each pattern vector pv_i in the historical data set, we can run the OSR algorithm using PVS-$\{pv_i\}$ as the initial data set and using the measurement vector composing pv_i as the measurement vector mv to be predicted. The pattern vector pv is removed from the data set in order to avoid any bias in the results. Thus, each time we run OSR, we know the actual value of the dependent variable we are trying to classify. This allows us to not only extract specific patterns for each pattern vector in the data set, but we are also able to classify each generated pattern as right or wrong at the time it is generated. The set of patterns generated through this iterative process forms a representation of the trends observable on this particular data set which we will call a Specific Pattern Set (SPS).

The SPS may be viewed as a hierarchical model (see Fig. 4) of the historical data set. Many of the patterns in the SPS 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 when the pattern is used) and *pattern reliability* (i.e., percentage of correct classification when the pattern is used) and *pattern reliability significance* (i.e., the probability that the observed reliability is greater than or equal to the one expected through a random classification by chance). These statistics can then be used to evaluate the pattern-based predictions as explained in the subsequent paragraphs. Thus, even though incomplete/partial information is available in the historical data set, accurate patterns may still be generated in some cases.

Recall that we assumed the patterns generated by OSR have the following ordered conjunctive normal form:

$$\text{Predicate}_1 \xrightarrow{\wedge} \text{Predicate}_2 \xrightarrow{\wedge} \cdots \xrightarrow{\wedge} \text{Predicate}_N$$

Also, recall the order in which the predicates appear is relevant in order to determine the contexts where they are relevant. A predicate is relevant only when the conditions defined by its preceding/parent predicates (i.e., the context of a predicate) are true.

Let Class Y_i be dependent variable class i. Let T be the number of generated pattern instances Pattern_j that predict Class Y_i. Let C be the number of pattern instances that *correctly* predicts Class Y_i (based on the actual Y value of the pattern vector for which the pattern was produced).

Then we define the *reliability* of Pattern_j with respect to the dependent variable class Class Y_i as:

$$R[\text{Class}\, Y_i;\, \text{Pattern}_j] = C/T$$

The probability that a pattern appears T times yielding a particular classification Class Y_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, $p = p(\text{Class } Y_i)$, i.e, the prior probability that the value of the dependent variable is in Class Y_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 *reliability significance (RS)* can be calculated using the following formula:

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

Example 12: For a given pattern, suppose that:

$C = 10$ (the number of times that the pattern was correct during the V-fold Cross Validation)

$T = 12$ (the total number of times the pattern was generated).

Also, suppose that there are exactly two dependent-variable classes and a uniform distribution in the historical data set, so that the prior probability of a pattern predicting each class is 0.5 for each dependent variable class.

Then, using the above formulas, this pattern has the following reliability and reliability significance.

$$R = 0.83$$

$$RS = 0.019$$

Since we are able to differentiate *significantly reliable* patterns from the nonsignificant and/or unreliable ones, we are able to know the reliability of a classification when we make it. That is, when we are trying to assess a new object, we run the OSR algorithm using that object as the measurement vector. This process extracts a set of patterns specific to that object. Then, when making a classification for this object, we know that a classification based on a reliable pattern with a sufficient level of significance (e.g., $RS < 0.05$) is believable, whereas one based on a reliable pattern with a poor level of significance is not.

Thus, our decision is based on the R's and RS's of each pattern in the hierarchy. Pattern reliability is used for classification while the variations in pattern entropy are used for interpretation. Although a reliable pattern always shows a low entropy, the opposite is not true (for reasons beyond the scope of this paper).

A poor reliability means that a pattern is not robust to "noise" (i.e., the dependent variable variations created by phenomena not captured by the pattern). A poor reliability significance may mean that the pattern is a result of noise (random variations) and not from a real trend in the data.

D. Support for Interpreting Patterns

As we have seen, patterns are useful for classifying variables of interest. However, more importantly, they are also useful in providing understandable/interpretable models. Patterns are much easier to interpret than regression coefficients. First of all, OSR takes into account interactions between explanatory variables, i.e., the fact that an explanatory variable can have a strong impact in a certain context and not be relevant in another one. These interactions do not have to be known before building the model as opposed to interaction terms in logistic regression [21]. Secondly, as we will see, an automated process (described below) can be defined to show strong associations that exist in a given context (this is needed to satisfy R4 of [12]). Finally, the variation in entropy generated by a particular predicate can help assess the significance of the impact of an explanatory variable (on the dependent variable) within a certain context. However, interpreting the raw patterns 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 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 patterns by assigning values to a small set of parameters.

As an example, let us define two patterns PT1 and PT2:

$$\text{PT1: Pred}_i \overset{\wedge}{\rightarrow} \text{Pred}_j$$

$$\text{PT2: Pred}_i \overset{\wedge}{\rightarrow} \text{Pred}_k$$

Suppose in the context where Pred_i is true, the pattern vector subset for which Pred_j is true happens to show a strong *association* with the one for which Pred_k is true. This implies that these predicates capture basically the same phenomenon. The strength of the association can be assessed by using normalized and symmetrical Chi-squared-based statistic such as Pearson's Phi [13]. A Chi-squared test can be performed 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, occurred randomly. This is a result of slight differences between the two predicates and therefore distinguishing between them does not help in the understanding of the object of study. This phenomenon is mainly due to complex interdependencies between X's that are often underlying the software engineering data sets.

In order to decide whether or not two strongly associated predicates should not be differentiated, the user declares a Phi value that 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* ($\text{Pred}_j \vee \text{Pred}_k$), implicitly meaning that its two component predicates are interchangeable in this context.

$$\text{PT}\{1, 2\}\text{: Pred}_i \overset{\wedge}{\rightarrow} (\text{Pred}_j \vee \text{Pred}_k)$$

Let us define a *composite predicate* to simply be a disjunction of predicates.

Example 13: Assume that in the context of a subsystem that has for focus data processing, most of the components with a large number of SLOC's are also the ones with a large Halstead's volume V. PT1 and PT2 will be merged if the level of association between the two second position predicates (who are in this case singletons) is higher than the Phi threshold defined by the user.

$$\text{PT1: SUBSYSTEM} \in \text{REAL-TIME CONTROL} \overset{\wedge}{\rightarrow} V \in \text{LARGE},\ R = 0.90,\ RS = 0.06$$

$$\text{PT2: SUBSYSTEM} \in \text{REAL-TIME CONTROL} \overset{\wedge}{\rightarrow} \text{SLOC} \in \text{LARGE},\ R = 0.92,\ RS = 0.07$$

$$\text{PT}\{1, 2\}\text{: SUBSYSTEM} \in \text{REAL-TIME CONTROL} \overset{\wedge}{\rightarrow} (V \in \text{LARGE} \vee \text{SLOC} \in \text{LARGE}) R = 0.91,\ RS = 0.01.$$

In this situation where PT1 and PT2 are both reliable but show a small number of occurrences in the specific pattern set (see previous section), then they will be associated with weak levels of significance (RS). merging them will increase this level of significance and keep the reliability (R) constant if the used Phi threshold is high enough.

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

In a similar manner, we can define a second merging principle. Suppose we have the same two patterns as defined above:

$$\text{PT1: Pred}_i \overset{\wedge}{\rightarrow} \text{Pred}_j$$

$$\text{PT2: Pred}_i \overset{\wedge}{\rightarrow} \text{Pred}_k.$$

However, this time suppose that Pred_j is the singleton predicate $X_1 \in \text{Class}_{km}$ and Pred_k is the singleton predicate $X_1 \in \text{Class}_{kn}$ where Class_{km} is a neighbor class of Class_{kn} (their boundaries may overlap). In this particular case, if the two patterns characterize subsets with no statistically significant difference in distribution on the dependent variable range, then they can be merged. This is because the variation from one class to the other seems to have a nonrelevant effect on the dependent variable under the context where Pred_i is true. Therefore, in order to assess if merging is possible, the probability that differences between distributions are 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 randomness. 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 tool developed to support the OSR approach, this is calculated through a binomial test for proportions.

Example 14: Assume that in the context of components with a large number of SLOC's and a large Halstead's volume V, the programmers experience of the programming language (ordinal factor on a scale 1–5) is a significant factor. Both PT1 and PT2 show a first position predicate that is the result of a previous merging according to the first principle presented above. Their second position predicate is similar but not identical. PT1 and PT2 will be merged into PT$\{1, 2\}$ if the level of similarity between the two second-position predicates (who are in this case singletons) is higher than the threshold defined by the user.

$$\text{PT1: } (V \in \text{LARGE} \vee \text{SLOC} \in \text{LARGE}) \overset{\wedge}{\rightarrow} \text{EXPERIENCE} \in [1, 2)$$

$$\text{PT2: } (V \in \text{LARGE} \vee \text{SLOC} \in \text{LARGE}) \overset{\wedge}{\rightarrow} \text{EXPERIENCE} \in [2, 3)$$

$$\text{PT}\{1, 2\}\text{: } (V \in \text{LARGE} \vee \text{SLOC} \in \text{LARGE}) \overset{\wedge}{\rightarrow} \text{EXPERIENCE} \in [1, 3)$$

Both of the merging principles defined above can be used simultaneously in order to obtain more significant and interpretable patterns. However,the merging process using both of them must be carefully defined. We have built a prototype tool where such mechanisms have been completely automated. A more precise definition of the pattern-merging algorithm is presented in Appendix B.

III. Validating the OSR Approach

In order to validate the OSR approach, we need to compare it to standard modeling processes that can be used for classification: logistic regression [21] and classification trees [27].

Our definition of a high-risk component is: any software procedure or function where errors were detected during system and acceptance test. Low risk is used to identify the remaining components of the system. In particular, we wish to build models that identify high-risk components for a particular category of errors: ones that characterize an incorrect reading or writing in a variable or a data structure.

A. Data Description

The data set was created using data collected from 146 components of a 260 KLOC Ada system. We selected randomly an equal number of both low- and high-risk components in the used data set. This was done in order to construct unbiased classification models. We selected all the high-risk components identified during test phases, and we randomly introduced an equivalent number of low-risk components among those available. A larger number of low-risk components in the data would lead *all modeling techniques* to generate more accurate models for the low-risk class and would provide mediocre models for the high-risk class (i.e., their results would not be representative of the actual capability of the models in terms of accurately identifying high-risk components).

The explanatory variables used to construct the models are static code and design metrics [2]–[4]. Others are well-known component-level complexity and size measures [5]. We will first summarize the architectural approach to measurement taken in this project and then define the assumptions on which the analysis was conducted.

The architectural view of the Ada system can be derived by identifying the major components of the system and determining the relationships among them. The library-unit aggregation (LUA), or the library unit and all its descendant secondary units [2], provides an interesting concept for an Ada system. Relationships between LUA's can include the importing relationship, or the relationship between an instantiation and its generic template. The increased use of Ada as a design as well as implementation language provides an opportunity to better assess the final product in its intermediate stages. Since the design and the final product are written in the same language we can use tools developed for analysis of Ada source code to provide an automated means for analyzing Ada designs. This automation is essential if one is to frequently measure and assess the design.

The metrics used in this study are derived from the architecture of the system and were obtained by an automated static analysis of the source code using the ASAP static-analysis program [17], UNIX utilities, and the SAS statistical-analysis system. At the heart of the measures are counts of declarations in an LUA—whether they are declarations made in the LUA, declarations imported to the LUA (i.e., declarations made in another LUA made visible by a "with" clause), declarations exported by the LUA (i.e., declarations made in the library unit and visible to other units that import the LUA), or declarations hidden from these importing units (i.e., declarations made in the body and subunits).

The collection of metrics was developed from hypotheses about the nature of the software-design process. These, in addition to other raw measures extracted from the source code, were used in this study. The metrics include ratios designed to indicate the extent of context coupling, visibility control, locality of imports, and parametrization. These characteristics are based on the following underlying assumptions:

- Assumption 1 (context coupling): Importing and/or exporting large amounts of declarations may require complex interfacing with the other LUA's of the system and is expected to be an error-prone factor.
- Assumption 2 (parametrization): The average number of parameters per program unit declaration in the LUA should have an impact on the probability of generating defects. The larger the parametrization of the LUA, the larger the number of abstractions to be dealt with, the greater the difficulty for a designer or a programmer to keep in memory his/her respective role, and the more complex it becomes to handle interaction with other LUA's.
- Assumption 3 (visibility control): The ratio of cascaded imports (declaration imports to a unit and whose visibility cascades to its descendent units [3] to direct imports in the LUA). This concept captures the extent to which declarations imported to where they are needed in the LUA. The larger the number of visible declarations unrelated to the problem addressed at a particular location in the LUA, the larger the risk of confusion or misunderstanding of those program abstractions.
- Assumption 4 (reuse): A high ratio of reused code in a LUA denotes the familiarity/understanding with the problem addressed and the computer-based solution, i.e., the LUA interface with other LUA's, its component interfact with other LUA's, its component interfaces, and its data structures. This is expected to lower the probability of defect.

In addition to the architectural metrics mentioned above, two main categories of component complexity metrics may be identified as well: size of the component and the structural or control-flow complexity of the component.

- Assumption 5 (component size): Different measures of size were used: the total number of Ada statements, the number of executable Ada statements and the number of source lines of code. Size measures have shown in the literature to be related to the probability of generating defects [22], [28].
- Assumption 6 (structural complexity): The structural complexity of the code should affect the probability of generating complex defects undetected during early walkthroughs and unit test.

B. Evaluating the Accuracy of the Models

We compare the results obtained using logistic regression and classification trees with those found using Optimized Set Reduction. The fully automated OSR process was used to generate the set of patterns partially presented in Section III.C. For each modeling approach, a V-fold cross-validation procedure was used [8]. Each pattern vector was successively removed from the data set. The model was built using the remainder of the data set and then used to predict the pattern vector extracted. The prediction is compared to the actual and this is repeated for each pattern vector in the data set. Unless the available data set is large, this validation method is preferable: this is an objective validation method (i.e., no arbitrarily selection of test sample) that allows model evaluations with a maximum number of observations.

The variable selection process used for building the regression models was a stepwise selection process with a predetermined selection criterion of $p = 0.05$. Dummy variables [16] were created to deal with discrete explanatory variables. Principal components [16], [21], [23] have been extracted and used in an attempt to optimize the accuracy of the regression models. Two regression models were built. The first one is based exclusively on the original explanatory variables. The second one uses, as explanatory variables, the generated principal components that are linear functions of the original explanatory variables, where each is orthogonal with respect to the others. With respect to classification trees, the algorithm provided by the S-PLUS system [27] was used and the parameters controlling the tree construction were tuned in order to get optimal accuracies.

When comparing modeling techniques with respect to identifying high-risk components, two different evaluation parameters must be considered simultaneously. Assume that when a high-risk component is identified, a remedial action is taken during the testing phase (e.g., more expensive and more effective code-reading technique) and that the benefit of this remedial action is validated and quantifiable. We

TABLE I
MODEL ACCURACIES

Model	Correctness	Completeness
Optimized Set Reduction	92.11% (70/76)	95.89% (70/73)
Classification trees	83.33% (60/72)	82.19% (60/73)
Logistic regression without Principal components	76.56% (49/64)	67.12% (49/73)
Logistic regression with Principal components	80.00% (52/65)	71.23% (52/73)

have to consider the *completeness* of the model (i.e., the percentage of high-risk components identified by the model). The benefit of this remedial action on the development process quality will be a function of completeness, since the larger the number of high-risk components identified, the higher the error-detection rate. Also, the *correctness* of the model (i.e., the percentage of components identified as high risk that are actually of high risk) allows the user to quantify the waste of resources due to the unnecessary applications of remedial actions.

Table I shows these two parameters for logistic regression, classification trees, and Optimized Set Reduction. OSR appears to be more accurate than both logistic regression and classification trees with respect to all the criteria considered. We conclude that the benefits of the remedial actions taken when identifying high-risk components are increased using OSR. These results seems to indicate an improvement of the OSR algorithm when compared with the earlier version presented in [10] where there was no significant accuracy differences when compared with logistic regression.

The results shown in Table I have been obtained following the classification rules below:

- Logistic regression: If the calculated probability of a component belonging to the high-risk class was below 0.5, the low-risk class was selected. Otherwise, the high-risk class was selected.
- Classification trees: The risk class was selected based upon the proportion of nonfaulty and faulty components in the matching tree leaf.
- OSR: For a given component, all the significantly reliable extracted patterns were considered for performing the classification. If those patterns all showed a high probability in the same risk class, then that class was selected. Otherwise, the risk class characterized by the pattern subset with the highest average pattern reliability was selected. If none of the extracted patterns happened to have a reliability significantly different from the random expected reliability, then the component was considered undetermined and thus classified randomly among the two risk classes.

By selecting biased classification rules (e.g., 0.4 decision boundary for logistic regression), the model completeness and correctness could be modified. However, when completeness increases, correctness decreases and vice versa. The best correctness/completeness trade off depends on the particular application of the model. The results in Table I were obtained using unbiased classification rules.

C. OSR Patterns' Interpretations

A comparison between the interpretability of logistic-regression equations and OSR patterns may be found in [11]. Issues associated with classification tree interpretation are discussed in [12]. In this section, we illustrate and evaluate the interpretability of OSR patterns. Some of the patterns characterizing high-risk components will be described to illustrate the interpretation process in the OSR context. Patterns will be presented in a format facilitating their readability. Class boundaries will not be shown since they are not meaningful to the reader. Instead their corresponding quantiles on the explanatory-variable range will be used to describe predicates.

1) Regression Equation: The regression equation generated is as follows:

$$\log\left(\frac{p}{1-p}\right) = 0.337 + 0.0103\,\mathrm{SLOC} - 0.00107\,\mathrm{LUADA} - 1.8274\,\mathrm{LUFREUC}$$

where $p =$ Prob(component is high risk).

One of the main problems of logistic regression models with respect to their interpretation is the inherent instability of regression coefficients when the underlying assumptions of the model are not met (see [11] for example and details). In some cases, looking at the correlation matrix may help avoid the problem when interpreting. Another related problem is that many good predictors were not selected by the stepwise selection process because of a strong correlation with already included parameters. In order to interpret the regression equations, the user has to look carefully at the correlation matrix and the regression equation in order to have some meaningful insight into the associations between explanatory variables and the dependent variable. Instability may be due to other causes like overinfluential data points (outliers) or interactions between explanatory variables [16], [21].

We will demonstrate in the next paragraphs that, on our data set, logistic regression does not extract much of the information provided by the data set. Some of the assumptions made in Section III.B.2 will be supported by the OSR patterns.

2) Patterns Characterizing High-Risk Components: The patterns listed below are the ones that seemed to confirm the assumptions stated in Section III.A. Generating interpretable patterns does not always imply generating easy-to-understand patterns because of the indirect and complex nature of some of the statistically significant associations extracted from our data sets. Moreover, since these statistical models do not deal with causality, interpretation becomes an even more sensitive process.

Patterns are grouped according to the assumption they support. For each pattern presented, the entropy associated with each predicate is shown just below the predicate itself. Patterns were generated entirely automatically without human intervention. As opposed to the classification-tree approach [27], no "tuning" of the algorithm was necessary since the parameters of the OSR algorithm are all intuitively meaningful (e.g., user-set statistical levels of significance for differentiating distributions) and can be set at once. The predicates' value intervals have been calculated automatically according to the procedure described in Section II.B.1. This approach

for handling predicate intervals automatically and dynamically (classes change in various contexts) gives more meaning to the interpretation of the OSR patterns. The first group of patterns is commented on in detail to remind the reader about how to read these patterns. A definition of the metrics appearing in the patterns presented below is provided in Appendix A.

Pattern Group 1: Complex code within a largely reused LUA (Assumptions 4 and 6)

NDMAX ∈ [52–100%]
$H = 0.89$
$\overset{\wedge}{\rightarrow}$ LUFREUS ∈ [0–71%] ⇒ High Risk
$H = 0.73$
NDMAX ∈ [52–100%]
$H = 0.89$
$\overset{\wedge}{\rightarrow}$ LUFREUC ∈ [0–81%] ⇒ High Risk
$H = 0.75$

Picking those components with a relatively small amount of reuse within the subset whose maximum statement nesting level is high implies a high probability that the component will be in the high-risk class.

The individual impact of predicates (here all singletons) on the risk (i.e., probability to be in the high-risk class) can be quantified by looking at the entropy variation they generate. NDMAX ∈ [52–100%] creates a variation of entropy of 0.11 (from 1.0, the initial set entropy, to 0.89). In this context, a variation of entropy of 0.16 can be observed for LUFREUS ∈ [0–71%] (from 0.89 to 0.73). However, there is no strong evidence that the amount of reuse in a LUA is a high-risk characteristic when NDMAX ∉ [52–100%]. In other words, this pattern group seems to indicate that architectural reuse pays off in terms of defect probability only in the context of complex components.

Pattern Group 2: Large compilation units within a LUA with a high level of parametrization (Assumptions 2 and 6).

(SLOC ∈ [57–100%] ∨ V ∈ [54–100%])
$H = 0.84$
$\overset{\wedge}{\rightarrow}$ LUPARPD ∈ [53–100%] ⇒ High Risk
$H = 0.46$

LUPARPD is an indicator of the average program-unit interface complexity within a particular LUA. This complexity seems even more difficult to handle for large components (i.e., large number of lines of code, operands, and operators). Based on the process defined in Section II.D, the reliability of this pattern has been assessed at 100% and appears to be significant at RS = 0.06. Since this data set is small, relatively few patterns show significances below 0.1. Here again, there is no strong evidence that LUPARPD is a high-risk characteristic in the context of small components. Large components with complex interfaces are risky while small components do not seem to be strongly affected.

Pattern Group 3: Large and complex compilation units within a LUA containing high quantities of cascaded imports (Assumptions 3, 5, and 6).

(SLOC ∈ [57–100%] ∨ V ∈ [54–100%])
$H = 0.84$
$\overset{\wedge}{\rightarrow}$ LUACTMAX ∈ [64–100%] ⇒ High Risk
$H = 0.0$
NDAV ∈ [65–100%]
$H = 0.92$
$\overset{\wedge}{\rightarrow}$ LUCMIMP ∈ [36–100%] ⇒ High Risk
$H = 0.0$

Importing large quantities of cascaded declarations seems to significantly increase the risk of defects even in the context of large and/or complex components, i.e., large number of lines of code, operands, and operators. Once again, small components do not seem to be affected.

In this pattern, the first predicate is an example of composite predicate and is the result of the merging process. Phi (i.e., the merging criterion) was fixed to 0.7 (one variable explains 70% of the variability of the other variable).

Pattern Group 4: Complex compilation units in the context of a LUA that exports/imports large quantities of declarations towards other LUA's (Assumption 1, 5, and 6).

LUWBYCU ∈ [79–100%]
$H = 0.78$
$\overset{\wedge}{\rightarrow}$ DOBJ ∈ [46–100%] ⇒ High Risk
$H = 0.34$
LUWBYCU ∈ [79–100%]
$H = 0.78$
$\overset{\wedge}{\rightarrow}$ VG ∈ [26–100%] ⇒ High Risk
$H = 0.44$
LUCC ∈ [93–100%] ⇒ High Risk
$H = 0.0$

This pattern group seems to indicate that interfacing with other compilation units in order to export complex compilation units (i.e., large number of declared/defined variables or a large cyclomatic complexity) shows a high defect risk. These patterns illustrate how the notion of context can play an important role when determining the impact of an explanatory variable. This shows that when one wants to validate assumptions, the answer may not be as simple as yes or no. In our particular example, most of the assumptions would not have been validated by simply looking at the regression model [14].

Pattern Group 5: When average-statement nesting level is high, the size of the component is large and this component has an ALgorithmic/COMPutational functionality (according to the NASA SEL taxonomy) [29], then there is a high probability that the component is high-risk. Note that this is an example

```
procedure MERGE (predicate tree, node, context, PHI, S)

  (1)  If (node is a terminal node of the predicate tree)
          then   RETURN

  (2) while (∃ cp_i, cp_j such that MNCP (cp_i , cp_j, S)) do
              UNION(predicate tree, node, cp_i, cp_j)

  (3) Calculate A_mxm^context

  (4) while (∃ cp_i, cp_j such that cp_i ~ cp_j) do

    . select cp_i and cp_j such that a_i,j^context is the strongest association in A_mxm^context
    . UNION(predicate tree, node, cp_i, cp_j)
    . recalculate A_m-1 x m-1^context, the association matrix for
          cp1, ..., cpi-1, cpi+1, ..., cpj-1, cpj+1,...,cpm, cp_iUj in context.

  (5) for each successor of node in predicate tree

    MERGE (predicate tree, successor, context ∧→ cp_node, PHI, S)

end MERGE
```

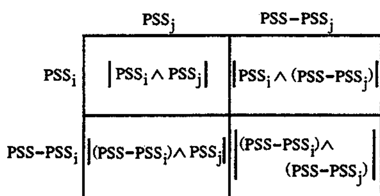
In step (4), a call is made to procedure UNION which is defined as follows:

```
procedure UNION (predicate tree, NODE, cp_i, cp_j)

   (1) Form a new node marked by the composite predicate cp_i U cp_j (i.e., cp_iUj)
   (2) Delete nodes marked by cp_i and cp_j under NODE
   (3) Combine all like subpaths rooted at cp_iUj

 end UNION
```

The merging process is initiated with the procedure call:

```
MERGE(predicate tree, root, Ø, PHI, S)
```

Algorithm B1.



	PSS_j	$PSS-PSS_j$
PSS_i	$\lvert PSS_i \wedge PSS_j \rvert$	$\lvert PSS_i \wedge (PSS-PSS_j) \rvert$
$PSS-PSS_i$	$\lvert (PSS-PSS_i) \wedge PSS_j \rvert$	$\lvert (PSS-PSS_i) \wedge (PSS-PSS_j) \rvert$

Fig. 5. Predicate association.

SPECIFIC PATTERN SET

$X_1 \in Class_1$ AND $X_3 \in Class_2$
$X_4 \in Class_1$ AND $X_2 \in Class_3$
$X_1 \in Class_1$ AND $X_3 \in Class_2$
$X_4 \in Class_1$ AND $X_5 \in Class_2$

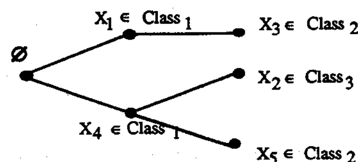


Fig. 6. Predicate tree.

of the use of non-singleton predicates.

$$\text{NDAV} \in [65\text{–}100\%]$$
$$H = 0.92$$
$$\overset{\wedge}{\rightarrow} (\text{ALCOMP YES} \wedge (\text{SLOC} \in [15\text{–}100\%] \vee$$
$$V \in [19\text{–}100\%] \vee \text{TOTASTMT} \in [23\text{–}100\%]))$$
$$H = 0.75$$

IV. Conclusions

Five main conclusions can be drawn from this paper:

1) Based on a rather small and incomplete data set, i.e., 146 Ada components, a completeness and a correctness above 90% has been obtained by using the OSR modeling process. If this level of accuracy is not sufficient, the user can tune the decisions boundary so he may increase either the correctness or completeness according to her/his specific needs.
2) OSR patterns appear to be more stable and interpretable structures than regression equations when their theoretical underlying assumptions are not met. Taking effective corrective actions is only possible when the impact of controllable factors on the parameters to be controlled (e.g., cost, quality) can be fully understood and quantified.
3) OSR patterns seem to generate a more complete set of information, i.e., validate more assumptions, than the logistic regression equation. This may be partially cor-

```
procedure DISCRETIZATION (dataset, EV, DV, criterion, classes)

   (1) sort dataset elements in increasing order according to elements' EV
   values
   (2) OPTIMAL_SPLIT(dataset, EV, DV, criterion, optimal_bound)

   (3) if(dataset has actually been split in (2))
         then {
                (3.1) update the definition of classes with newly calculated
                optimal bound
                (3.2) extract two subsets sset1, sset2 of dataset where
                EV < optimal_bound and EV > optimal_bound,
                respectively
                (3.3) DISCRETIZATION (sset1, EV, DV, criterion, classes)
                (3.4) DISCRETIZATION (sset1, EV, DV, criterion, classes)
               }

end DISCRETIZATION

The procedure for splitting datasets may be defined as follows:

procedure OPTIMAL_SPLIT (dataset, EV, DV, criteria, optimal_bound)


for all data vectors Vi in dataset (in sorted order)
   {
     Case 1: there is a change in DV value but not in EV value
        { homogeneous = FALSE }

     Case 2: there is a change in EV value (from EVV1 to EVV2) and while EV
     values remained constant and equal to EVV1, homogeneous remained equal to
     TRUE
       {
         /*
         STEP1: calculate entropy of the distribution on the DV range for the
         dataset subset lying in the interval strictly below EVV2 (SSET2)

         STEP2: Calculate the level of significance of the difference in
         distribution between dataset and SSET2.

         STEP3: If the the level of significance is below criterion and the
         entropy is below the minimal entropy calculated so far, then
         optimal_bound is assigned with EVV2
         */

         Entropy2 = H(SSET2, DV)
```

Algorithm B2 (part 1 of 2).

rected by looking at the explanatory variable correlation matrix. However, this is an extremely tedious and not always helpful task, e.g., issues like interactions between explanatory variables are still not addressed.

4) OSR classifications were found to be more accurate than logistic regression equations. This also confirms previous studies showing similar results for other kinds of applications [11], [12]. Therefore, the Optimized Set Reduction approach seems to be a good alternative and/or complement to multivariate logistic regression in this application domain.
5) OSR classifications were found to be more accurate than a classification tree. This also confirms earlier result we obtained on the data sets used in [11], where classification trees were performing poorer than both logistic regression and OSR. These results seem to suggest that the classification tree structure, even though simple to generate and use, might be too simplistic for modeling complex artifacts such as high-risk software components.

From a more general perspective, the OSR approach is a data analysis framework that successfully integrates statistical and machine-learning approaches to empirical modeling with respect to specific software engineering needs. It provides support for dealing with both partial information and model interpretation and is not based on a severely constraining set of hypotheses.

Appendix A
Definitions of the Metrics Appearing in the Paper

- **Library Unit Aggregation (LUA) metrics:**
  - LUCATMAX: total number of cascaded program unit declarations/maximum possible number of cascaded program unit declarations
  - LUCMIMP: cascaded imported program unit declarations/direct imported program unit declarations
  - LUWBYLU: number of library unit aggregations that contain a with statement to this compilation unit
  - LUWBYCU: number of compilation units that contain a with statement to this compilation unit
  - LUPARPD: number of parameters per program unit declaration in the LUA

```
    s2 = DIFFDIST(dataset, SSET2, DV)
    if(Entropy2 < H(dataset, DV) and s2 < criterion)
     then optimal_bound = EVV2

  }

Case 3: there is a change in EV value (from EVV1 to EVV2) and while EV
values remained constant and equal to EVV1, DV values changed at least once
and homogeneous = FALSE
  {
    /*
    SSET1 is the dataset subset lying in the interval strictly below EVV1.

    STEP1: calculate entropy of the distribution on the DV range for the
    dataset subset lying in the interval strictly below EVV1 (SSET1)

    STEP2: Calculate the level of significance of the difference in
    distribution between dataset and SSET1.

    STEP3: If the the level of significance is below criterion and the
    entropy is below the minimal entropy calculated so far, then
    optimal_bound is assigned with EVV1

    STEP4: repeat same procedure as above for SSET2

    STEP5: set homogeneous to TRUE
    */

    Entropy1 = H(SSET1, DV)
    s1 = DIFFDIST(dataset, SSET1, DV)
    if(Entropy1 < H(dataset, DV) & Entropy1< optimal_bound & s1 < criterion)
     then optimal_bound = EVV1
    Entropy2 = H(SSET2, DV)
    s2 = DIFFDIST(dataset, SSET2, DV)
    if(Entropy2 < H(dataset, DV) & Entropy2 < optimal_bound & s2 < criterion)
     then optimal_bound = EVV2
    homogeneous = TRUE;
  }

Case4: no change in DV value

/* Do nothing */

} /* end of for loop */

end OPTIMAL_SPLIT
```

Algorithm B2 (part 2 of 2).

- LUFREUC: fraction of old (reused verbatim) number of components in the LUA
- LUFREUS: fraction of old (reused verbatim) number of SLOC's in the LUA
- LUADA: number of ADa statements in the LUA
- LUCC: unique Imported declarations/unique exported declarations

- **Compilation unit metrics:**
  - NDMAX: maximum statement nesting level
  - NDAV: average statement nesting level
  - SLOC: source lines of code
  - V: Haldstead's volume
  - VG: cyclomatic complexity
  - DOBJ: number of declared variables

Appendix B
Algorithms

The Merging Algorithm: This merging process can be formalized using the following definitions and algorithms: Recall the definition of predicate and composite predicate from Sections II.A.1 and II.D. Let *cp* represent a composite predicate. Then, we define:

Definition A1: A *context* (C) is an ordered conjunction of composite predicates that defines a subset of pattern vectors PSS (i.e., PSS = SUBSET(PVS, C)).

Definition A2: An *association coefficient* a_{ij}^C is an assigned statistical degree of association between cp_i and cp_j in a data set PSS = SUBSET(PVS, C). Let PSS_i = SUBSET(PSS, cp_i) and let PSS_j = SUBSET(PSS, cp_j).

A two row-two column contingency table is defined as shown in Fig. 5. Based on this table, a Chi-square-based statistic (Person's Phi), the degree of association between cp_i and cp_j in PSS is calculated and assigned to a_{ij}^C. Note that this association coefficient is calculated in the context of C (i.e., PSS = SUBSET(PVS, C)) and therefore is only valid under C.

Definition A3: An *association matrix* $A_{n\times n}^C$ is a square matrix of association coefficients calculated under a context C, where the rows/columns are marked by composite predicates, i.e.,

$$A_{n\times n}^C \text{ contains all } a_{ij}^C, \qquad i, j \in \{1, \cdots, n\}$$

Definition A4: Two composite predicates cp_i and cp_j are said to be similar in the context of C if $a_{ij}^C \geq$ PHI (the minimal level of association defined by the user). This association will be denoted as $cp_i \approx cp_j$.

Definition A5: A *predicate tree* is a tree representation of the patterns generated when extracting the Specific Pattern Set (SPS) process. As mentioned is Section II.D, the SPS is 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 SPS. Each path of a predicate tree represents a pattern (see Fig. 6).

Note that in the example, all of the predicates are singleton. This could represent a predicate tree that summarizes an OSR run. During the merging process, branches will be merged and composite predicates created at the nodes.

Definition A6: Two composite predicates cp_i, cp_j are said to be "mergable neighboring composite predicates" if the following conditions are fulfilled:

1) There exist two predicates Pred_m and Pred_n, where $\text{Pred}_m = (X_i \in \text{class}_{ik})$ and $\text{Pred}_n = (X_i \in \text{Class}_{it})$ (both are singleton predicates) such that Pred_m and Pred_n are each disjuncts in cp_i and cp_j, respectively.
2) Class_{ik} and Class_{it} are neighboring (or overlapping) classes on variable X_i domain.
3) cp_j and cp_j yield similar distributions on the dependent variable range (i.e., the level of significance of the two distributions being different is above S (user defined)).

If these three conditions are true, then MNCP (cp_i, cp_j, S) is TRUE.

We can now define the merging algorithm (Algorithm B.1).

Discretization Algorithm (Algorithm B2). The algorithm's parameters are defined as follows:

- EV: the explanatory variable whose range is going to be discretized
- DV: the dependent variable of the model to be built
- data set: set of pattern vectors to be discretized along the scale of EV.
- criterion: maximum level of significance accepted to recognize two distributions as different
- classes: the definition of the intervals (classes) on EV's range, i.e., a set of pairs of boundaries.

ACKNOWLEDGMENT

The authors would like to thank W. Agresti, F. McGarry, and J. Valett for their support in providing some of the data used in this analysis. Also, we would like to thank S. Morasca and W. Thomas for their numerous comments that helped improve both the content and the form of this paper.

REFERENCES

[1] A. Agresti, *Categorical Data Analysis.* New York: Wiley, 1990.
[2] W. Agresti, W. Evanco, and M. Smith, "Early experiences building a software quality prediction model," in *Proc. Fifteenth Ann. Software Eng. Workshop*, Nov. 1990.
[3] W. Agresti and W. Evanco, "Projecting software defects from analyzing Ada design," *IEEE Trans. Software Eng.*, vol. 18, no. 11, Nov. 1992.
[4] W. Agresti, W. Evanco, D. Murphy, W. Thomas, and B. Ulery, "Statistical models for Ada design quality," in *Proc. Fourth Software Quality Workshop* (Alexandria Bay, New York), Aug. 1992.
[5] V. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Commun. ACM*, vol. 27, no. 1, Jan. 1984.
[6] V. Basili, "Quantitative evaluation of software methodology," in *Proc. First Pan Pacific Comput. Conf.* (Australia), July 1985.
[7] V. Basili and H. Rombach, "The TAME project: towards improvement-oriented software environments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, June 1988.
[8] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees.* Monterey, CA: Wadsworth & Brooks/Cole, 1984.
[9] L. Briand and A. Porter, "An alternative modeling approach for predicting error profiles in Ada systems," *EUROMETRICS '92* (European Conference on Quantitative Evaluation of Software and Systems, Brussels, Belgium), Apr. 1992.
[10] L. Briand, V. Basili, and C. Hetmanski, "Providing an empirical basis for optimizing the verification and testing phases of software development," *IEEE Int. Symp. Software Reliability Eng.* (North Carolina), Oct. 1992.
[11] L. Briand, W. Thomas, and C. Hetmanski, "Modeling and managing risk early in software development," *Int. Conf. Software Eng.* (Maryland), May 1993.
[12] L. Briand, V. Baili, and W. Thomas, "A pattern recognition approach for software engineering data analysis," *IEEE Trans. Software Eng.*, vol. 18, no. 11, Nov. 1992.
[13] D. Card and W. Agresti, "Measuring software design complexity," *J. Syst. Software*, vol. 8, no. 3, Mar. 1988.
[14] J. Capon, "Statistics for the social sciences," Wadworth Publishing Co., 1988.
[15] J. Cendrowska, "PRISM: an algorithm for inducing modular rules," *J. Man-Machine Studies*, vol. 27, p. 349.
[16] W. Dillon and M. Goldstein, *Multivariate Analysis: Methods and Applications.* New York: 1984.
[17] D. Doubleday, "ASAP: an Ada static source code analyzer program," Tech. Rep. TR-1895, Dept. Comput. Sci., Univ. of Maryland, Aug. 1987.
[18] W. Evanco and W. Agresti, "Statistical representations and analyses of software," in *Proc. 24th Symp. Interface of Computing Sci. Statistics* (College Station, Texas), Mar. 1992.
[19] J. Gannon, E. Katz, and V. Basili, "Measures for Ada packages: an initial study," *Commun. ACM*, vol. 29, no. 7, July 1986.
[20] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Software Eng.*, vol. SE-7, no. 5, Sept. 1981.
[21] D. Hosmer and S. Lemeshow, *Applied Logistic Regression.* New York: Wiley, 1989.
[22] R. Michalski, "Theory and methodology of inductive learning," in *Machine Learning* (vol. 1), R. Michalski, J. Carbonell, and T. Mitchell, Eds. Los Altos, CA: Morgan Kaufmann.
[23] J. Munson and T. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Software Eng.*, vol. 18, no. 5, May 1992.
[24] H. Potier, J. Albin, R. Ferreol, and A. Bilodeau, "Experiments with computer software complexity and relaibility," in *Proc. Sixth Int. Conf. Software Eng.*, Sept. 1982.
[25] J. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, 1986.
[26] H. D. Rombach, "A controlled experiment on the impact of software structure on maintainability," *IEEE Trans. Software Eng.*, vol. SE-13, no. 3, Mar. 1987.
[27] J. Chambers and T. Hastie, "Statistical models in S." Pacific Grove, CA: Wadsworth & Brooks/Cole.
[28] R. Selby and A. Porter, "Learning from examples: generation and evaluation of decision trees for software resource analysis," *IEEE Trans. Software Eng.*, vol. 14, no. 12, Dec. 1988.
[29] Software Engineering Laboratory, NASA Goddard Flight Center, "Data collection procedures for the software engineering laboratory database," TR SEL-92-002, Mar. 1992.

Lionel C. Briand received the Masters degree in computer science from the Université Pierre et Marie Curie (Paris 6) in 1988. He is currently working toward the Ph.D. degree at the Université Paris-sud, Laboratoire de Recherche en Informatique.

He was a software engineer at CISI Ingenierie Co., Paris, France, working on databases, expert systems, and software-design methodologies. He is presently a Faculty Research Assistant in the Department of Computer Science of the University of Maryland, College Park and a member of the Software Engineering Laboratory (NASA-GSFC, U. of MD, CSC). His research interests include software measurement, evaluation, and risk management for large-scale software development.

Victor R. Basili (M'83–SM'84–F'90) is a Professor in the Institute for Advanced Computer Studies and the Computer Science Department of the University of Maryland, College Park, where he served as chairman for six years. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, Boeing, NRL, NSWC, and NASA. He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering, and quality assurance by developing models and metrics for the software-development process and product. He has authored over 100 papers.

Dr. Basili was the General Chairman of the 15 International Conference on Software Engineering in May, 1993 and serves on the editorial board of the *Journal of Systems and Software*. He has been Editor-in-Chief of the IEEE Transactions on Software Engineering and Program Chairman for several conferences, including the 6th International Conference on Software Engineering in Japan. In 1982, he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING for his paper on the evaluation of methodologies. He was one of the recipients of the NASA Group Achievement Award for the GRO Ada experiment in 1989 and the NASA/GSFC Productivity Improvement and Quality Enhancement Award, for the Cleanroom project in 1990. He is a member of the Computing Research Board and a former Governing Board member of the IEEE Computer Society.

Christopher J. Hetmanski received the B.S. degree in both mathematics and computer science from Loyola College, Baltimore, MD, in May, 1991. He received the M.S. degree, with high honors, in computer science from the University of Maryland, College Park, in May, 1993. While at the University of Maryland, Chris worked as a Research Assistant in the Software Engineering Laboratory in areas such as software process modeling, software metrics and system-reliability optimization.