SOFTWARE ENGINEERING LABORATORY SERIES

SEL-93-001

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME XI

NOVEMBER 1993

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1993	Contractor Report

4. TITLE AND SUBTITLE

Collected Software Engineering Papers:
Volume XI

5. FUNDING NUMBERS

552

6. AUTHOR(S)

Software Engineering Laboratory

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland

8. PERFORMING ORGANIZATION REPORT NUMBER

SEL-93-001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CR-189347

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified–Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document is collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from November 1992 thorugh November 1993. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 11th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

14. SUBJECT TERMS

Software Models, Software Measurement, Technology Evaluations

15. NUMBER OF PAGES

93

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

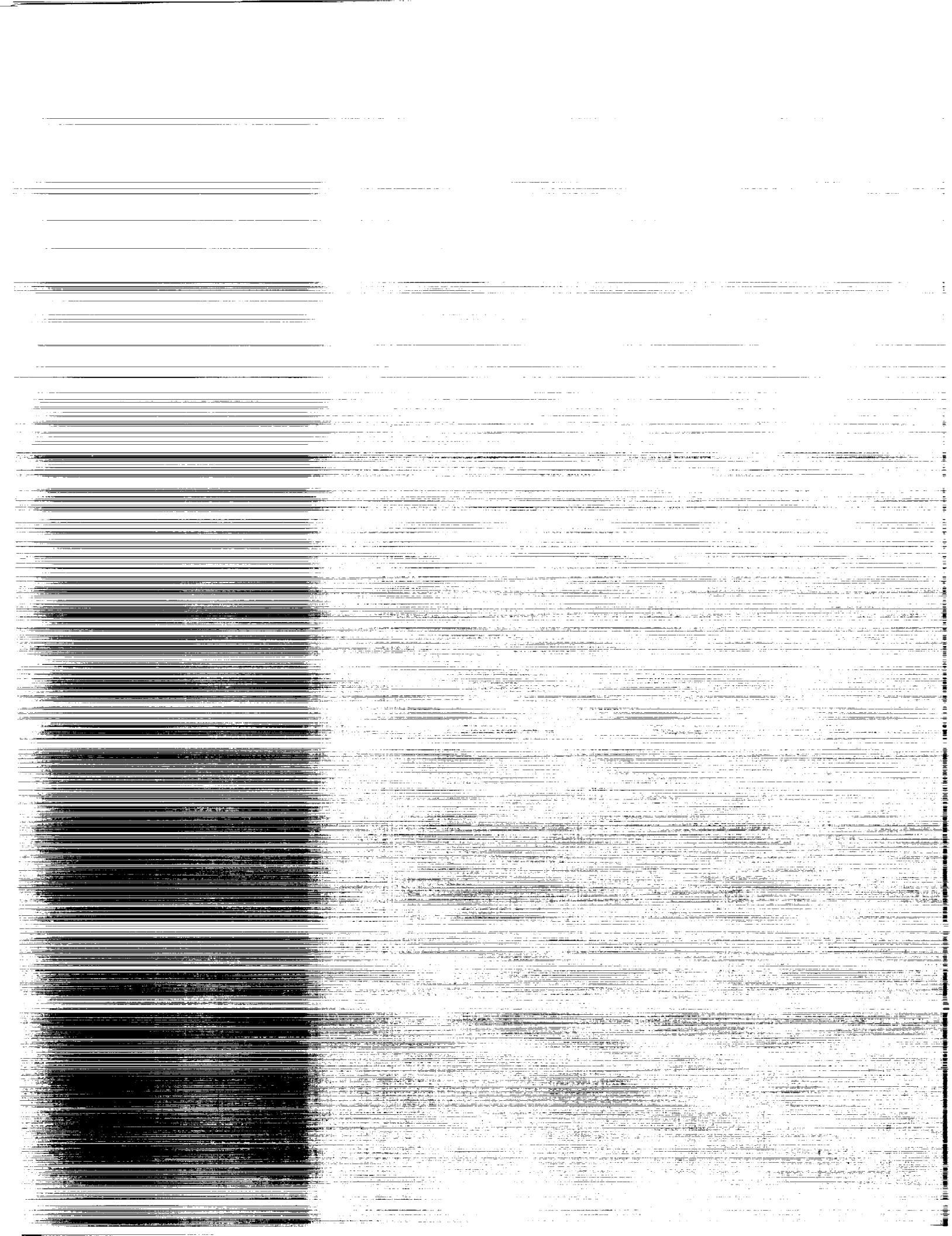
The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION



SECTION 1—INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from November 1992 through November 1993. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 11th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

For the convenience of this presentation, the five papers contained here are grouped into three major sections:

- Software Models
- Software Measurement
- Technology Evaluations

The first section (Section 2) includes studies on models for managing risk early in the software development process, for identifying high-risk software components, and for estimation and prediction in software management. Section 3 presents a study on the measurement of software maintainability during the design phase. A study on the impacts of object-oriented technologies in the SEL appears in Section 4.

The SEL is actively working to understand and improve the software development process at Goddard Space Flight Center (GSFC). Future efforts will be documented in additional volumes of the *Collected Software Engineering Papers* and other SEL publications.

SECTION 2 - SOFTWARE MODELS

SECTION 2—SOFTWARE MODELS

The technical papers included in this section were originally prepared as indicated below.

- *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, L. C. Briand, V. R. Basili, and C. J. Hetmanski, TR-3048, University of Maryland, Technical Report, March 1993
- "Modeling and Managing Risk Early in Software Development," L. C. Briand, W. M. Thomas, and C. J. Hetmanski, *Proceedings of the Fifteenth International Conference on Software Engineering* (ICSE 93), May 1993
- "An Information Model for Use in Software Management Estimation and Prediction," N. R. Li and M. V. Zelkowitz, *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components[1]

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

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 predictiing 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.

Key words: *Optimized Set Reduction, data analysis, fault-prone Ada components, stochastic modeling, machine learning, classification trees, logistic regression.*

1 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 [BP84, SP88, MK92]. Therefore, if we can identify components that are 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 is minimal, incomplete and heterogeneous [BBT92]. 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.

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 [Agr90, HL89]. This technique has been applied to software engineering modeling [MK92], as well as other experimental fields, and will therefore be used as a baseline for comparison in this paper.

1 Research for this study was supported in part by NASA grant NSG 5123 and NSF grant 01-5-24845

Many assumptions and constraints inherent to this technique make it difficult to apply in a software engineering context: (1) non-monotonicity of the probability density function on the explanatory variable range (2) interactions between explanatory variables are difficult to take into account when performing exploratory data analysis with numerous explanatory variables.

• Classification trees, which are described in [BF+84]. They were used to address software engineering modeling issues in [PA+82, SP88]. A review may be found in [CE87, BBT92]. Their strengths stem 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 non-relevant and non-significant information in the tree.

• Optimized Set Reduction (OSR), which has been developed at the University of Maryland [BBT92] in the framework of the TAME project [BR88] and has already been applied to several software engineering applications [BBT92, BBH92, BTH93]. It is partially based on both machine learning principles [Q86, BF+84], and univariate statistics [Cap88]. 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 [BBT92]. OSR generates logical expressions which represent patterns in a data set. For instance, consider the following example of a simple pattern (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 would be: NONLOC_IMP = High $\wedge$ (NESTING = High $\wedge$ GLOBALS = 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 2, 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 [BBT92]) 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 [BBT92]) 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 which were 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 [BBT92]).

Also in contrast to [BBT92], 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 [BBH92] based on the analysis of FORTRAN systems).

In Section 3, 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 3.2 we compare the generated OSR models to both logistic regression and classification tree models with respect to their accuracy. In Section 3.3, we discuss the interpretability of the OSR models. Finally, in Section 4, we outline the main conclusions of this paper and define the future directions of the research.

2 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 Xs). 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 Xs 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 Xs may be: complexity metrics, system architecture metrics or developer related evaluation of skills.

2.1 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 2.1.1 presents the notions informally to provide the reader with some intuition about the method. The rest of Section 2 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.

2.1.1 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 = $\{X_1, X_2, ..., X_m\}$). We define the term *pattern vector* to mean one of these such experiences. Assume the dependent variable's value domain (dom(Y)) is divided into a set of disjoint and exhaustive classes which can be either intervals (if the Y is continuous) or categories (if the Y is discrete). Each explanatory variable has its own value domain ($dom(X_i)$) which, like dom(Y) is divided into a set C of disjoint and exhaustive value classes $C = \{Class_{i1}, Class_{i2}, ... Class_{ik}\}$. 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 $MV = \underset{i \in (1 .. m)}{\times} dom(X_i)$. Likewise, the pattern vector value domain (i.e., the domain of the vectors in the data set) can be represented as $PV = dom(Y) \times MV$. 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 $Class_{ij} \subseteq dom(X_i)$. A singleton predicate (also written $X_i \in Class_{ij}$) 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 which are built upon 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 [BBT92]).

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 a 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.

2.1.2 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(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[M83] in order 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.

$$H(PSS, Y) = - \sum_{ClassY_j \subseteq C} p(PSS, ClassYj) \log_{|C|} p(PSS, ClassYj)$$

where,

- . PSS is a set of pattern vectors
- . $ClassY_j$ is a class defined on dom(Y)
- . $p(PSS, Y_j)$ is the prior probability that a vector which is an element of PSS has a dependent variable value belonging to the dependent variable class Y_j

• **Definition 2:** DIFFDIST(PSS_i, PSS_j, Y)

DIFFDIST(PSS_i, PSS_j, Y) = 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 [BBT92]. 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).

• **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*.

$PSS \subseteq PVS \wedge mv \in MV \wedge \exists Pred$ such that $(\forall pv \in PSS$, IS_TRUE(Pred , pv) $\wedge$ IS_TRUE(Pred , mv)) $\Rightarrow$ VALID(PSS, mv)

• **Definition 4:** EMIN(PSS, PSS_j, Y)

EMIN(PSS, PSS_j, Y) = 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, $H(PSS_j, Y) \leq H(PSS_k, Y)$. 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.

$PSS \subset PVS \wedge PSS_j \subset PSS \wedge$ (DIFFDIST(PSS_j, PSS, Y) $\wedge$ ($\forall PSS_k \subset$ PSS, $k \neq j$, DIFFDIST(PSS_k, PSS, Y) $\wedge H(PSS_j, Y) \leq H(PSS_k, Y)$)) $\Rightarrow$ EMIN(PSS, PSS_j, Y)

• **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 which 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 Figure 1, based upon a given measurement vector (*mv*) and a given historical dataset, the optimal subset extracted by OPT and characterized by the predicate on the left hand side of Figure 1 indicates a strong probability for Y to lie in the interval Y2. This may be used for predicting the class 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.

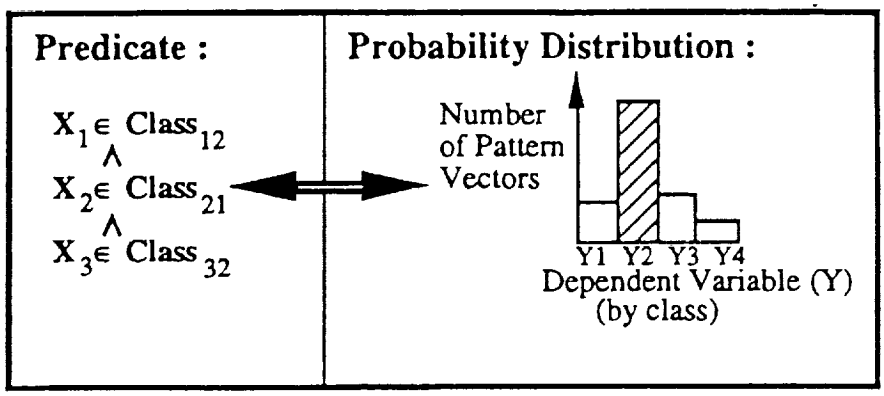


Figure 1: Classification with Extracted Subsets

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

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

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 Figure 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 which addresses these issues, discussing its relationship to the function OPT. This is the Optimized Set Reduction process which can roughly be described by a three step recursive algorithm where entropy is optimized in a stepwise manner.

2.2 The OSR Algorithm

The goal of the OSR algorithm is to produce a set of *patterns* which 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 upon 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 2.2.2, we shall describe the OSR algorithm in detail. However, before doing so, we need to define a number of preliminary concepts that are used in the algorithm.

2.2.1 Preliminary Definitions

• **Definition 6:** OSR Pattern

As mentioned above, OSR generates patterns. A pattern is an *ordered* conjunction of predicates which 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 and does not have any logical impact on the characterization of optimal subsets We will call the ordered expression to the left of a given predicate in a pattern the *context* of the predicate. This addresses issue number 4 mentioned above.

Example 7: Define two predicates

$Pred_1$ = SUBSYSTEM $\in$ REAL-TIME CONTROL $\wedge$ SUBSYSTEM $\in$ LARGE
$Pred_2$ = #GLOBAL VARIABLES $\in$ LARGE.

If we assume the pattern $Pred_1 \overset{\wedge}{\rightarrow} Pred_2$ was generated by OSR, we can see that this pattern characterizes a pattern vector set suggesting a high *risk* which is defined, in this particular example, as the probability of detecting errors that are difficult to correct during the test phases (see Figure 2).

This pattern ($Pred_1 \overset{\wedge}{\rightarrow} Pred_2$) has a 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.)

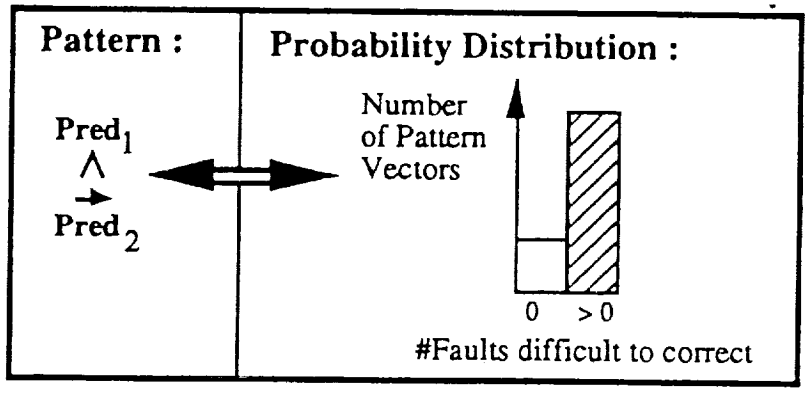


Figure 2: Classification Using Patterns

• **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 an exhaustive and disjoint set of classes ($Class_{i1}$ ... $Class_{ik}$ 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 [BBT92] which states that an explanatory variable may be a 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) [DG84]. However, none appear 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 II.

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 that are 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 non-monotonic 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. Figure 3 illustrates the output of the algorithm. We assume an actual underlying and unknown non-monotonic probability density function and an observed sequence of Y values on the explanatory variable X range. We also assume two classes (1, 2) are defined on the Y value domain . Using the DISCRETIZE algorithm produces Boundary1 and Boundary2 in Figure 3, which creates the corresponding set of three explanatory variable value classes across the X range.

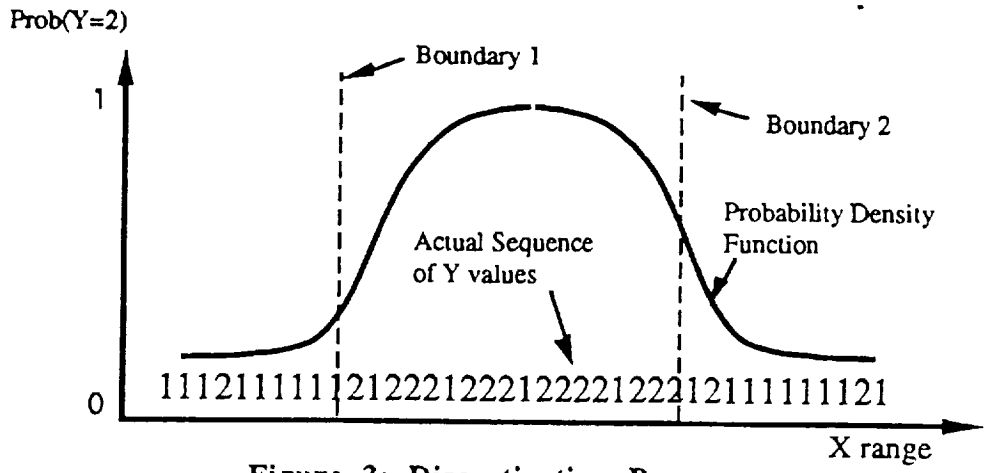


Figure 3: Discretization Process

• **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: $\{X_i \in Class_{i1}, ..., X_i \in Class_{ik}\}$. GENERATE_SINGLETONS(PSS, mv) generates the set of all singleton predicates SP such that $SP = \{Pred_i \mid IS\_TRUE(Pred_i, mv)\}$.

• **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.

$PSS \subseteq PVS \wedge (H(SUBSET(PSS, Pred), Y) < H(PSS, Y) \wedge$
$DIFFDIST(PSS, SUBSET(PSS, Pred), Y)) \Rightarrow SIG\_PREDICATE(PSS, Pred, Y)$

> **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 which is 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. A binomial test for proportions [Cap88] is used to assess the significance of the observed difference in entropy.

• **Definition 10:** MINIMAL(PSS, $Pred_i$, Y)

The predicate $Pred_i$ is said to be *minimal* for the pattern vector set PSS if it characterizes a subset of PSS which shows a significantly different distribution across the Y classes and there exists no other predicate $Pred_j \Rightarrow Pred_i$ such that $Pred_j$ characterizes a subset of PSS which 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*.

$SIG\_PREDICATE(PSS, Pred_i, Y) \wedge (\forall j, Predj \Rightarrow Pred_i, j \neq i, (\neg SIG\_PREDICATE(PSS, Pred_j, Y)) \Rightarrow MINIMAL(PSS, Pred_i, Y)$

Example 9: Assume that the predicate $Pred_1$ = SUBSYSTEM ∈ REAL-TIME CONTROL ∧ SUBSYSTEM ∈ 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 ∈ 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 a set of predicates which 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 which are conjuncts of the singletons in SP and which 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.

$$\text{VALID\_PREDICATES}(PSS, PRED_c, SP, Y) = \{Pred_i \mid \text{MINIMAL}(PSS, Pred_i, Y) \wedge Pred_i \subseteq SP \wedge (\forall j, Pred_j \in PRED_c, \forall X \text{ such that } X \in \{ X_k \mid X_k \in EV \wedge \text{MEMBER}(X_k, Pred_j)\}, \neg\text{MEMBER}(X, Pred_i) \}$$

• **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.

$$\text{EXTRACT\_SUBSETS}(PSS, PRED) = \{PSS_i \mid Pred_i \in PRED \wedge PSS_i = \text{SUBSET}(PSS, Pred_i)\}$$

2.2.2 The Algorithm

When the dependent variable's value domain is defined on a continuous 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 dataset 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 an integer range from 0 to 5, 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:

ClassY1 = Y in [0, 1) Low Risk Components
ClassY2 = Y in [1, +∞) 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, ∅, 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:

$PSS \subset PVS \wedge mv \in MV \wedge \exists Pred_i$ such that $(\forall pv \in PSS$, IS_TRUE($Pred_i$, pv) $\wedge$ IS_TRUE($Pred_i$, mv) $\wedge$ **MINIMAL(PSS, $Pred_i$, Y)**) $\Rightarrow$ VALID(PSS, mv, Y)

Then, assuming the definition of OPT uses this new definition of VALID, we can then define OSR in the following way:

$$OSR(PSS, mv, PRED_C, Y) = \left\{ \begin{array}{l} \bigcup_{PSS_i \in Opt(PSS, mv, Y)} (OSR(PSS_i, mv, PRED_C \cup Pred_i, Y)), \text{ if } OPT(PSS, mv, Y) \neq \emptyset \\ \{PSS\}, \text{ otherwise.} \end{array} \right\}$$

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. Thus, if we implicitly order the paths, the ordered conjunction of predicates along each recursive path is a pattern (see Definition 6

and Figure 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 figure 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 which 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:

$$\text{NUM\_IMPORTS} = \text{HIGH} \overset{\wedge}{\rightarrow} (\text{NESTING=HIGH} \wedge \text{CMPLX=HIGH})$$

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).

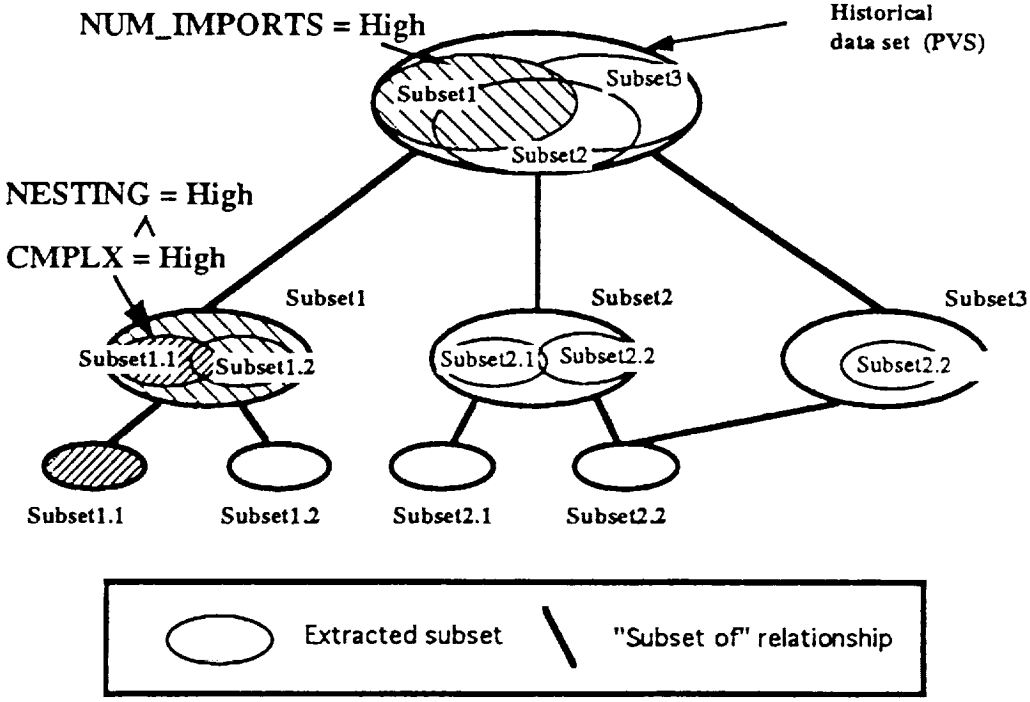


Figure 4: An Example of OSR Hierarchy

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* [BBT92] 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.

2.3 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* [BF+84]. For each pattern vector *pv* in the historical data set, we can run the OSR algorithm using *PVS - {pv}* as the initial data set and using the measurement vector composing *pv* as *mv*. 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 figure 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 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:

$$\text{Predicate1} \overset{\wedge}{\rightarrow} \text{Predicate2} \overset{\wedge}{\rightarrow} \ldots \overset{\wedge}{\rightarrow} \text{PredicateN}$$

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 ClassY_i be dependent variable class i. Let T be the number of generated pattern instances *Pattern*$_j$ that predict ClassY_i. Let C be the number of pattern instances which *correctly* predict ClassY_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 ClassY_i as:

$$R[\text{ClassY}_i ; \text{Pattern}_j] = C / T$$

The probability that a pattern appears T times yielding a particular classification ClassY_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 $\text{Class}Y_i$.

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 variables classes and an 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 non-significant 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 process 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).

Note: a poor reliability means that a pattern is not robust to "noise" (i.e., the dependent variable variations created by non-measured phenomena). A poor reliability significance may mean that the pattern is a result of noise or more complex phenomena resulting from the OSR process (again beyond the scope of this paper.

2.4 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 [HL89]. Secondly, as we will see, a process (described below) can be defined to show strong associations that exist in a given context (this is needed to satisfy R4 of [BBT92]). 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 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: $Pred_i \overset{\wedge}{\rightarrow} Pred_j$

PT2: $Pred_i \overset{\wedge}{\rightarrow} Pred_k$

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 Chi-squared based statistic such as Pearson's Phi [CA88]. 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 Xs 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 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* ($Pred_j \vee Pred_k$) implicitly meaning that its two component predicates are interchangeable in this context.

$$PT\{1,2\}: Pred_i \overset{\wedge}{\rightarrow} (Pred_j \vee Pred_k)$$

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

Example11: Assume that in the context of a subsystem that has for focus data processing, most of the components with a large number of SLOCs 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.

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

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

PT{1,2}: SUBSYSTEM $\in$ REAL-TIME CONTROL $\overset{\wedge}{\rightarrow}$ (V $\in$ LARGE $\vee$ SLOC $\in$ 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:

PT1: $Pred_i \overset{\wedge}{\rightarrow} Pred_j$

PT2: $Pred_i \overset{\wedge}{\rightarrow} Pred_k$

However, this time suppose that $Pred_j$ is the singleton predicate $X_l \in Class_{km}$ and $Pred_k$ is the singleton predicate $X_l \in Class_{kn}$ 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 non-relevant 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.

Example12: Assume that in the context of components with a large number of SLOCs 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 which 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.

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

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

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

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 II.

3 Validating the 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 [HL89], classification trees [S92].

Our definition of a high risk component (procedure or function) is: any software component 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.

3.1 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 models more accurate for the low risk class and would therefore 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. Some of these metrics are taken from a project whose goals were to build multi-variate models of software quality based on architectural characteristics of Ada designs [AES90,AE92,AE+92]. Others are well known component level complexity and size measures[BP84]. We will first summarize the architectural approach to measurement taken in this project and then define the assumptions upon 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 [AES90], provides an interesting concept for an Ada system. Relationships between LUAs 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 [Dou87], 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 were developed from hypotheses about the nature of the software design process and further details can be found in [AES90,AE92,AE+92]. 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 amount 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 difficultly for a designer or a programmer to keep in memory their respective role, the more complex it becomes to handle interaction with others LUA's.

• Assumption 3 (Visibility control): The ratio of cascaded imports (declaration imports to a unit and whose visibility cascades to it's descendent units[AE+92]) to direct imports in the LUA . This concept captures the extent to which declarations are 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 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 [SP88, MK92].

• 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.

3.2 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 3.3. For each modeling approach, a V-fold cross validation procedure was used [BF+84]. Each pattern vector was successively removed from the dataset. The model was built using the remainder of the dataset 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 dataset. Unless the available dataset is large, this validation method is preferable: this is an objective validation method (i.e., no arbitrarily selection of test sample) that allow 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 [DG84] were created in order to deal with discrete explanatory variables. Principal components [DG84, HL 89, MK92] 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 which 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 [S92] was used and the parameters controlling the tree construction were tuned in order to get optimal accuracies. However, this process was quite tedious since no guideline or rational exists for tuning these parameters despite the great instability of the generated trees.

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 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 1 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 [BBH92] where there was no significant accuracy differences when compared with logistic regression.

The results shown in Table 1 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 non-faulty 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 tradeoff depends on the particular application of the model. The results below were obtained using unbiased classification rules.

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)

Table 1: Model Accuracies

3.3 OSR Patterns' Interpretations

Comparison between the interpretability of logistic regression equations and OSR patterns may be found in [BTH93]. Issues associated with classification tree interpretation are discussed in [BBT92]. In this section, we illustrate and evaluate the interpretability of OSR patterns. Some of the patterns characterizing "data value / structure" errors will be described in order 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 (in the appropriate context) will be used to describe predicates.

3.3.1 Regression Equation

The regression equation generated is as follows:

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

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 [BTH93] 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 [DG84, HL89].

We will demonstrate in the next paragraphs that, on our dataset, logistic regression does not extract a lot of the information which is provided by the data set. Some of the assumptions made in 3.2.2 will be supported by the OSR patterns.

3.3.2 Patterns for Data Value / Structure Errors

The patterns listed below are the ones that seemed to confirm the assumptions stated in section 3.1. Our goal was not to make assumptions based on the generated patterns since this is a risky and dangerous approach to data analysis, i.e., exploratory data analysis. As a matter of fact, many of the generated patterns were not clearly understandable to us and did not fit in our list of assumptions. Generating interpretable patterns does not imply generating easy to understand patterns, which is due to the indirect and complex nature of some of the statistically significant associations extracted from our data sets. Moreover, since 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 (here singleton predicates) is shown just below the predicate itself. Patterns were generated entirely automatically without human intervention. As opposed to the classification tree approach [S92], no "tuning" of the algorithm was necessary since the parameters of the OSR algorithm are all intituively 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 2.2.1. 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 in detail in order 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 I.

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

$$\underset{H = 0.89}{NDMAX \in [52\% - 100\%]} \overset{\wedge}{\rightarrow} \underset{H = 0.73}{LUFREUS \in [0\% - 71\%]} \Rightarrow \text{High Risk}$$

$$\underset{H = 0.89}{NDMAX \in [52\% - 100\%]} \overset{\wedge}{\rightarrow} \underset{H = 0.75}{LUFREUC \in [0\% - 81\%]} \Rightarrow \text{High Risk}$$

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 (i.e., to generate errors).

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 $\in$ [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 $\in$ [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 $\notin$ [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).

$$\underset{H = 0.84}{(SLOC \in [57\% - 100\%] \vee V \in [54\% - 100\%])} \overset{\wedge}{\rightarrow} \underset{H = 0.46}{LUPARPD \in [53\% - 100\%]} \Rightarrow \text{High Risk}$$

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 2.4, 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).

$$\underset{H = 0.84}{(SLOC \in [57\% - 100\%] \vee V \in [54\% - 100\%])} \overset{\wedge}{\rightarrow} \underset{H = 0.0}{LUACTMAX \in [64\% - 100\%]} \Rightarrow \text{High Risk}$$

$$\underset{H = 0.92}{NDAV \in [65\% - 100\%]} \overset{\wedge}{\rightarrow} \underset{H = 0.0}{LUCMIMP \in [36\% - 100\%]} \Rightarrow \text{High Risk}$$

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.

• **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).

$$\underset{H = 0.78}{LUWBYCU \in [79\% - 100\%]} \overset{\wedge}{\rightarrow} \underset{H = 0.34}{DOBJ \in [46\% - 100\%]} \Rightarrow \text{High Risk}$$

$$\underset{H = 0.78}{LUWBYCU \in [79\% - 100\%]} \overset{\wedge}{\rightarrow} \underset{H = 0.44}{VG \in [26\% - 100\%]} \Rightarrow \text{High Risk}$$

$$LUCC \in [93\% - 100\%] \Rightarrow \text{High Risk}$$
$$H = 0.0$$

This pattern group seems to indicate that interfacing with other compilation units in order to export complex compilation unit (i.e., large number of declared / defined variables or a large cyclomatic complexity) shows a high defect risk. These patterns illustrates 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 [CAP88].

• **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), then there is a high probability that the component is high risk. Note that this is an example of the use of non-singleton predicates.

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

4 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 the 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 corrected 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 [BBT92, BTH93]. 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 results we obtained on the datasets used in [BTH93] 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 components.

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

5 Acknowledgments

We would like to thank William Agresti, Frank McGarry and Jon Valett for their support in providing the data used in this analysis. Also, we would like to thank Sandro Morasca and William Thomas for their numerous comments that helped improve both the content and the form of this paper.

References

[Agr90] A. Agresti, *Categorical Data Analysis*, John Wiley & Sons, 1990.

[AES90] W. Agresti, W. Evanco, and M. Smith, "Early Experiences Building a Software Quality Prediction Model", *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November, 1990.

[AE92] W. Agresti and W. Evanco, "Projecting Software Defects from, Analyzing Ada Designs", *IEEE Trans. Software Eng.*, 18 (11), November, 1992 (to appear).

[AE+92] W. Agresti, W. Evanco, D. Murphy, W. Thomas, and B. Ulery, "Statistical Models for Ada Design Quality", *Proceedings of the Fourth Software Quality Workshop*, Alexandria Bay, New York, August, 1992.

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

[Bas85] V. Basili, "Quantitative Evaluation of Software Methodology", *Proceedings of the First Pan Pacific Computer Conference*, Australia, July 1985.

[BR88] V. Basili and H. Rombach,"The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

[BF+84] L. Breiman, J. Friedman, R. Olshen and C. Stone, *Classification and Regression Trees,* Wadsworth & Brooks/Cole, Monterey, California, 1984.

[BP92] 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, April 1992.

[BBH92] L. Briand, V. Basili and C. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development", *IEEE International Symposium on Software Reliability Engineering*, North Carolina, October 1992.

[BTH93] L. Briand, W. Thomas and C. Hetmanski, "Modeling and Managing Risk early in Software Development", International Conference on Software Engineering, Maryland, May 1993

[BBT92] L. Briand, V. Basili and W. Thomas, "A Pattern Recognition Approach for Software Engineering Data Analysis", *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[CA88] D. Card and W. Agresti, "Measuring Software Design Complexity", *Journal of Systems and Software*, 8 (3), March, 1988.

[Cap88] 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: Methods and Applications*, Wiley and Sons, 1984.

[Dou87] D. Doubleday, "ASAP: An Ada Static Source Code Analyzer Program", TR-1895, Department

of Computer Science, University of Maryland, August, 1987.

[EA92] W. Evanco and W. Agresti, "Statistical Representations and Analyses of Software", *Proceedings of the 24th Symposium on the Interface of Computing Science and Statistics*", College Station, Texas, March, 1992.

[GKB87] J. Gannon, E. Katz, and V. Basili, "Measures for Ada Packages: An Initial Study", *Communications of the ACM*, 29 (7), July, 1986.

[HK81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Trans. Software Eng.*, 7 (5), September, 1981.

[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.

[MK92] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs", *IEEE Trans. Software Eng.*, 18 (5), May, 1992.

[PA+82] H. Potier, J. Albin, R. Ferreol and A. Bilodeau, "Experiments with Computer Software Complexity and Reliability", *Proceedings of the Sixth International Conference on Software Engineering* , September, 1982.

[Qui86] J. Quinlan, "Induction of Decision Trees", *Machine Learning* 1, Number 1, 1986.

[Rom87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Trans. Software Eng.*, 13 (3), March, 1987.

[S92] J. Chambers, T. Hastie, "Statistical Models in S", Wadsworth & Brooks/Cole Advanced Books & software, Pacific Grove, California

[SP88] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis", *IEEE Trans. Software Eng.*, 14 (12), December, 1988.

Appendix I: Definitions of the metrics appearing in the paper

- **Library Unit Aggregation (LUA) metrics:**

  . LUACTMAX: 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

  . 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: Halstead's volume

  . VG: cyclomatic complexity

  . DOBJ: number of declared variables

Appendix II: 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 section 2.1.1 and 2.4. 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 Figure 5.

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

Figure 5: Predicate Association

Based on this table, a Chi-Square based statistic (Pearson'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.

example: $A_{n \times n}^{C}$ contains all a_{ij}^{C}, $i,j \in \{1,...,n\}$

• **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

$cp_i = cp_j$.

• **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 2.4, 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 represent a pattern (see Figure 6)

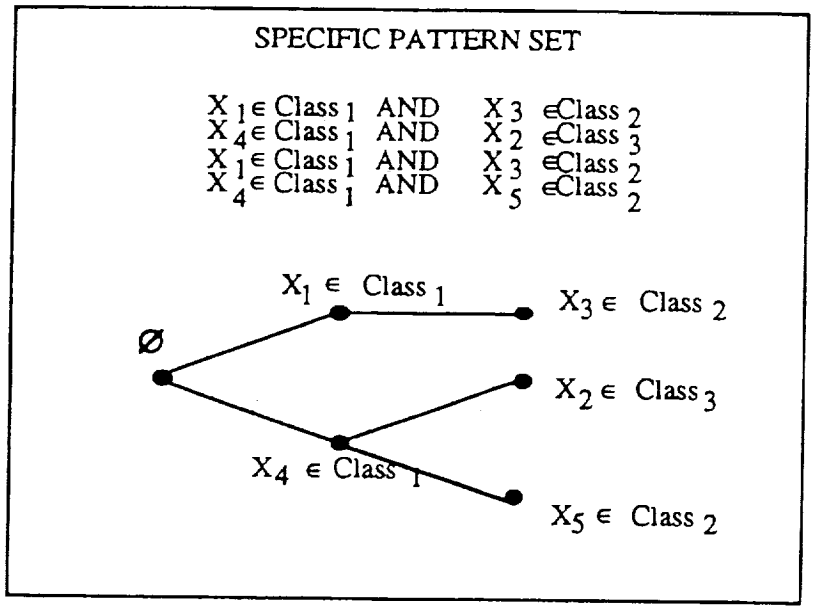


Figure 6: Predicate Tree

Note that in the above example, all of the predicates are singleton. This could represent a predicate tree which 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 $Pred_m = (X_i \in class_{ik})$ and $Pred_n = (X_i \in Class_{it})$ (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_i 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 as follows:

```
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
          cp_1, ..., cp_i-1, cp_i+1, ..., cp_j-1, cp_j+1, ..., cp_m, 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, ∅, PHI, S)
```

Discretization Algorithm

Procedure parameter definitions:

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

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

   (1) sort dataset elements in increasing order according to elements' variable
   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
                variable < optimal_bound and variable > 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)
```

```
        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
```

Modeling and Managing Risk Early in Software Development*

Lionel C. Briand, William M. Thomas† and Christopher J. Hetmanski
Department of Computer Science
University of Maryland, College Park, MD 20742

Abstract

In order to improve the quality of the software development process, we need to be able to build empirical multivariate models based on data collectable early in the software process. These models need to be both useful for prediction and easy to interpret, so that remedial actions may be taken in order to control and optimize the development process. We present an automated modeling technique which can be used as an alternative to regression techniques. We show how it can be used to facilitate the identification and aid the interpretation of the significant trends which characterize "high risk" components in several Ada systems. Finally, we evaluate the effectiveness of our technique based on a comparison with logistic regression based models.

1 Introduction

It is often noted that a small number of software components are responsible for a large part of the difficulty during software development. In light of this relationship, there have been a number of studies that focus on the development and use of models to identify these "high risk" components [PA+82, SP88, BP92, MK92]. There are two different aspects to be treated when one builds a risk model. First, metrics that are good predictors of risk should be defined and validated. Second, a suitable (in terms of underlying assumptions) modeling technique should be used so that prediction is accurate and interpretation possible. Once these "high risk" components have been identified, the development process can be optimized to reduce risk. This can be performed from various perspectives of risk, e.g, number of errors, error density, associated cost of change during either testing or maintenance. For example, additional testing can be applied to those components that have been determined to be likely to contain a high density of defects.

Process improvement in terms of the prediction of defects in the delivered product is one area that has received a significant amount of attention recently [SP88, MK92]. Recent studies have focused on the identification of problem areas during the design phase, noting that the software architecture is a major factor in the number of errors and rework effort found in later phases [HK81, ROM87, CA88, AES90]. If such potential problem areas can be detected during the design, as opposed to during implementation or test, the development organization may have more options available to mitigate the risk. For example, rather than intensively testing the "problem components", one might restructure the system to avoid the potential problems entirely. While this may be an option during the design phase, it is a very unlikely scenario late in the implementation phase.

Thus our goal is to use measures of the design phase to determine potential problem areas in the delivered product, and allow for a wide range of preventive/corrective actions to be taken. Examples of these types of actions include increasing testing, providing additional documentation, re-designing a part of the system, and providing additional training.

We need a modeling process that will allow for the reliable detection of potential problem areas and for the interpretation of the cause of the problem so that the most appropriate remedial action can be taken. In this context, we will examine the use of the following modeling approaches:

- Logistic regression, which is one of the most common classification techniques [Agr90]. This technique has been applied to software engineering modeling [MK92], as well as other experimental fields.

- Optimized Set Reduction (OSR), which is based on both statistics and machine learning principles [Qui86]. This approach has been developed at the University of Maryland and has been applied in several software engineering applications [BBT92, BBH92].

*This work was supported in part by NASA grant NSG-5123
†Also with The MITRE Corp., McLean, VA.

Both techniques will be evaluated with respect to their accuracy, constraints of use and ease of interpretation. In summary, we intend to show that OSR may be used as an alternative to logistic regression to generate models using architectural metrics which can be used to control a software development project. Through the interpretation of OSR model, we will demonstrate how OSR can be useful in performing exploratory data analysis. Also, we will show that OSR models will allow one to predict and explain, at an early stage, where and why difficulties are likely to occur within the system architecture. Thus, planning, managing resources and quality control can become more effective.

This paper presents the results of an investigation into the use of the two different modeling techniques to support the identification and understanding of high risk components in Ada designs. Section 2 will define the notion of components that we used in this study of Ada systems, identify what we had targeted as "high risk", and present an overview of the modeling techniques. Section 3 will present the architectural metrics that were used in the study, and describe the underlying principles on which they are based. Section 4 presents the predictive accuracy of each technique, while section 5 discusses and provides interpretations of the models. Section 6 presents the major conclusions of the study.

2 Experiment Design and Modeling Techniques

2.1 Objectives of the Study

The data used in our analysis originates in the NASA/Goddard Space Flight Center Flight Dynamics Division. A number of Ada systems have been built in this environment, and a wealth of data has been collected on their development, ranging from items such as component reuse, to error origins, and to the amount of effort spent performing various development activities.

A research project at the MITRE corporation studied a number of these Ada systems, and related characteristics of software architecture to quality factors concerning the presence of defects, the difficulty in correcting defects, and the difficulty in adapting the system to changes [AES90, EA92]. Several regression-based models have been developed to predict quality factors from architectural characteristics. These models tackled issues such as identifying error-prone or difficult to modify components. We defined the notion of a high risk component based on a combination of the above two quality factors. We defined two classes, *high isolation cost*, and *high completion cost*, and built models for each. From change report form data, a component would be placed in the *high isolation cost* class if there is a defect in the component that requires more than one day to isolate and understand. Similarly, a component was placed in the *high completion cost* class if there is a defect associated with it that required more than one day to complete the error correction, once it had been isolated. The reason for the use of these two models is to better understand the major influences in error isolation difficulty and error completion difficulty, which are likely to be different. These definitions of high cost components provide a more useful notion of difficulty to a project manager. The statement "there is likely to be defect associated with this component, and its going to be hard to fix", is a much stronger statement than "there is likely to be defect associated with this component."

A random selection of approximately 150 components from three Ada systems was used to calibrate and evaluate the two modeling techniques (logistic regression and OSR). Our notion of a "component" is described in section 3. An equal number of components were chosen from the two classes in order to ensure the construction of unbiased models and thereby facilitate their evaluation and comparison. For each component (X) in the sample, a model was developed based on the remaining components ({Sample - X}) and used to "predict" whether the component (X) is likely to be in the high risk class. This model validation method, known as V-fold cross-validation [BF+84], is commonly used when data sets are small.

Characteristics of the design were used as explanatory variables in order to build classification models of the Ada components. These design characteristics are identified in section 3 along with our definition of software "component". The classification models will identify the components where at least one error was detected during system and acceptance test such that the error isolation/correction effort required more than one day. Two modeling approaches were evaluated: logistic regression with a stepwise variable selection, and optimized set reduction. The characteristics of each technique are briefly described in the following paragraphs.

2.2 Logistic Regression

The first technique, logistic regression analysis, is based on the following relationship equation [Agr90]:

$$\log\left(\frac{p}{1-p}\right) = C_0 + C_1 * X_1 + C_2 * X_2 + \ldots + C_N * X_N$$

As an example, we can assume P to be the probability of a component to be in the high risk class, i.e. is likely to have at least one difficult error to correct., and the X_i's to be the design metrics included as predictors in the model. In the two extreme cases, i.e. when a variable is either non significant or differentiates entirely the two

classes, the curve (between P and any X_i) approximates a horizontal line and a vertical line respectively. In between, the curve takes a S shape. However, since P is unknown, the coefficients C_i will be approximated through a likelihood function optimization. Based on the equation above, the likelihood function of a data set of size D is:

$$L = \prod_{i=1}^{D} \frac{e^{(C_0+C_1*X_{1i}+...+C_n*X_{ni})*Y_i}}{1+e^{(C_0+C_1*X_{1i}+...+C_n*X_{ni})*Y_i}}$$

The coefficients that will maximize the likelihood function will be the regression coefficient estimates. However, this expression is not maximizable through analytical methods and therefore numerical algorithms (e.g. Newton-Raphson) are used to maximize L. A heuristic stepwise process for explanatory variable selection can be used to build the model.

Two main problems were met while using this approach:

(1) Several of the design metrics are ratios and many instances show zero denominators and therefore undefined values. Logistic regression cannot gracefully handle "undefined" cases. For instance, in this case, using "dummy" variables would increase the number of explanatory variables by nearly fifty percent. Therefore, in order to address the problem, we replaced the undefined values with zeros and calculated the coefficients from this modified data set. In this way, we insure that undefined instances will not affect the calculation of the likelihood function.

(2) Two of the dependent variables are defined on nominal scales. The only solution to deal with such variable is to use "dummy" variables [DG84]. In this case, the two nominal variables force us to generate eight dummy variables to be considered during the stepwise variable selection process. For a larger number of symbolic/nominal variables, this issue may become a serious handicap for using the logistic regression approach.

Before starting the stepwise logistic regression process, it is possible to reduce the dimensionality of the sample space (i.e. 68 explanatory variables in our case) by performing a principal component analysis [DG84] on the available design and size metrics. Thus, we hope to be able to extract a smaller number of variables capturing most of the variation observed in the sample space. As shown by [DG84, MK92], this may increase the stability of the stepwise variable selection process and therefore improve the predictive result of the regression model.

2.3 Optimized Set Reduction

The second approach, Optimized Set Reduction (OSR), is described in [BBT92, BBH92]. It is a modeling approach which is based on both statistical and machine learning principles [BSOF84]. Given an historical data set, OSR automatically generates (through a search algorithm) a collection of logical expressions referred to as *patterns* which characterize the trends observable in the data set. As an example of a pattern, consider the following:

(Predicate$_k$ OR Predicate$_l$) AND Predicate$_m$ => Risk_class$_i$.

where predicates have the form ($EV_i \in EVclass_{ij}$), meaning that a particular explanatory variable EV_i belongs to part of its value domain, i.e. $EVclass_{ij}$.

The expression on the left hand side of "=>" characterizes a set of components from the historical data set. For example, if a given component is such that it makes the left hand side of the above logical expression true, it implies that it is likely to be in the Risk_class$_i$. For each pattern generated by OSR, a reliability of prediction (i.e. estimated probability of performing the right classification) and a its statistical significance (how likely is this probability due to chance?) are calculated based on the learning set. When using OSR, a collection of relevant patterns associated with a component are identified based on the learning set (i.e. the data set minus the component). As a result of this process, it is possible that several patterns characterizing the same component could yield contradictory classifications. In this case, the conflict is solved by first eliminating patterns that do not show a significant reliability. Then, if the remaining patterns are still in conflict, the pattern that shows the highest reliability is used for the classification.

Patterns provide interpretable models where the impact of each predicate can be easily evaluated. When interpreting patterns, they should be read as regular logical expressions with one main exception: the order of the terms on each side of the "AND" operator is meaningful. A predicate on a right-hand side of an AND operator is statistically significant (i.e. has a significant impact on the risk class probabilities) only if the predicates on the left-hand side are true. In our example, (Predicate $_k$ OR Predicate $_l$) is significant independent of any context while Predicate$_m$ is only significant in the context where (Predicate $_k$ OR Predicate $_l$) is already true. Strong associations (as defined by the user) between predicates are visible through OR connections.

The OSR process will generate a set of patterns specific of the data set provided. However, interdependencies

between explanatory variables may cause OSR to produce numerous similar patterns which capture essentially the same phenomena. This presents two problems (1) it can make pattern interpretation more confusing, since it masks predicate associations in various contexts, and (2) it hides the significance of phenomena which are represented by several similar patterns whose statistical significance appears weak independently, but is quite significant when grouped together. In order to address these issues, algorithms, supported by tools, have been designed to merge "similar" patterns according to a user defined, statistically based, degree of similarity [BBH92]. These algorithms have been used in order to obtain the patterns presented in the next sections.

It should be noted that in the design of OSR, we have alleviated some of the problems encountered in the logistic regression model. The "division by zero" cases can be handled as well as any other cases since it is simply defined as just another class of the variable's domain. Also, nominal and continuous explanatory variables are selected and included in the model in a consistent manner, since both are considered as predicates. One possible limitation of OSR is that it requires continuous explanatory variable ranges to be divided in intervals. However, this is done automatically by clustering algorithms which calculate optimal boundaries.

2.4 Evaluation of Models

Accuracy of models is compared from two different perspectives: their *completeness* and their *correctness* in identifying high risk components. Completeness is the percentage of components that have generated difficult errors that have been actually recognized as such by the model. It tells us how effective the model is in determining the high risk components, and thus can be used to determine the benefit of applying remedial actions to these components. Correctness is the percentage of correct classifications when a component is classified in the high risk class. It tells us the cost of achieving that level of effectiveness in the model. Both measures are necessary to perform a cost/benefit analysis on remedial actions taken on the components identified as high risk. For instance, given a particular completeness, if correctness is low, the remedial action will be taken on many components which are actually not high risk, creating waste of resources and therefore increasing the cost of the action. On the other hand, if correctness is high, waste of resources will be minimized.

Interpretability of a model will be defined as "the capability, based on the model, to quantify in various contexts the association (interpretable as a cause-effect relationship) of explanatory variables with the defined notion of risk". This will be assessed for each modeling technique by evaluating their capability to provide such quantification.

3. Metrics Used in the Study

The metrics used in the study were obtained from a project whose goals were to build multivariate models of software quality based on architectural characteristics of Ada designs [AES90]. This project explores the view that characteristics of the software architecture can be extracted from Ada designs using static analysis, and can be used to predict various quality factors in the delivered product [AE92,AE+92,EA92].

3.1 An Architectural View of the System

The increased use of Ada as a design as well as an implementation language offers the opportunity to better assess the product in its intermediate stages. Since the design and the final product are written in the same language, Ada, 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 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 [AES90], has been noted as providing an interesting view of an Ada system. Example relationships between LUAs are the importer/exporter relationship and the relationship between an instantiation and its generic template. Characteristics of the LUAs and the relationships between LUAs were used to develop multivariate statistical models of quality factors such as defect density, error correction effort, and change implementation effort [AE92, AE+92, EA92]. The characteristics that were included in this study are described below.

3.2 Description of Design Characteristics

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 [Dou87], UNIX utilities, and the SAS statistical analysis system. They were generated as part of a research project performed at the MITRE Corporation whose goal was to develop models to predict various product qualities throughout the development process [AES90,AE92]. 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. those declared in another LUA made visible by a "with" clause), declarations exported by the LUA (i.e. declarations made in the LUA , and visible to other units that import the LUA), and declarations hidden from these importing units (i.e. declarations made in the associated body and subunits). A collection of metrics were 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 indications of design characteristics such as the extent of imports, context coupling, visibility control, locality of imports, and parameterization. These characteristics are explained below.

- Imports: the number of declarations imported (via a "with" clause) to a LUA. This measure is an indication of the amount of services used by a particular unit. A unit that does not import must develop hidden units to allow for the provision of services listed in its specification. On the other hand, a unit that imports too extensively may not be cohesive as possible. At times, either extreme may be a problem area.

- Context Coupling Ratio: the ratio of declarations imported by a LUA divided by the declarations exported by the LUA. This measure is an indication of the amount of services used by a particular unit relative to what services it provides. As with imports, either extreme may be a problem area.

- Locality of Imports: the percentage of imported declarations that originate from the same subsystem as the LUA of interest. It is believed that a developer is more familiar with LUAs of the same subsystem as the LUA that he is developing. When the LUA imports primarily from these "local" LUAs, there may be a reduced chance of a misunderstanding about the imports.

- Parameterization: This characteristic relates to how well the LUA is parameterized. The metric used is the average number of parameters per program unit declaration in the LUA. Too many parameters may be an indication that the unit is not cohesive, and thus could be more difficult to understand, while too few may result in an inflexible structure, and thus make adaptation and modification more difficult. Either extreme may adversely affect quality.

- Visibility Control: This design characteristic attempts to capture the extent to which declarations are imported to where they are needed, as suggested in [GKB86]. The metric used is a ratio of "cascaded imports" (or declarations directly imported to a higher level unit in the LUA, and whose visibility "cascades" to the descendent units), to direct imports [AES90]. When this ratio is equal to one, it indicates that declarations are being imported directly to each compilation unit that uses them. As the ratio increases, it indicates the extent of indirect import visibility, relative to direct import visibility, which can be taken as a proxy for whether the imports are occurring only at the level in which they are needed.

3.3 Measurement of Design Characteristics

The above design characteristics have only been described in a general manner. Different ways of counting declarations will result in a collection of similar metrics. For example, the ratio of imports over exports can be defined in terms of total declarations (i.e. the total number of imported declarations divided by the total number of exported declarations), or in terms of subprogram declarations (i.e. the number of imported subprograms divided by the number of exported subprograms). While these are two different measures, there is a significant degree of similarity. However, one major difference is that the count of all subprogram declarations should be available at an earlier phase of the design than the count of total declarations. Thus a model using metrics based on subprogram declarations can be can be applied at an earlier stage in the design than one using metrics based on total declarations. The metrics used are distinguished by differentiating between various types of declarations, (i.e. packages, subprograms, tasks, types, subtypes, objects, formal parameters, constants, and exceptions), and by whether they differentiate overloaded names. Counts of declarations made in each LUA, as well as the metrics described in 3.2, were also used in the analysis.

4 Classification Accuracy of the Generated Models

4.1 Classification Rules

As said in section 2.3, during the OSR process, several patterns are generated for each LUA to be predicted. For each of these patterns, a specific classification is calculated based upon the pattern vector subset that it characterizes and its corresponding distribution across risk classes. Those classifications are used in order to determine the final classification of the LUA. Unfortunately, the patterns may yield different classifications. In this case, the first criterion used for classifying the LUA is the pattern reliabilities. The pattern with the maximum reliability is selected for classification. When several patterns show an identical reliability, then the statistical significance of this reliability (i.e. probability that this reliability is obtained by chance) is compared. The pattern with the

best level of significance is selected. With respect to logistic regression, the calculated risk class probabilities (see section 2.2) are used. A decision boundary of 0.5 was used since the original data set contained the same number of data points within each risk class, i.e. the a priori class probabilities are 0.5.

4.2 Predictive Accuracy

Tables 1 and 2 compares the modeling techniques, for both high isolation cost and high completion cost, respectively, the average correctness (i.e. the percentage of correct classifications in both high and low risk classes), the correctness of the model when looking at the high risk class only, and the completeness of the model with respect to the high risk class LUAs.

	Logistic Regression	OSR
Completeness	62%	84%
High Class Correctness	83%	83%
Average Correctness	75%	82%

Table 1: High Isolation Cost Model Accuracies

	Logistic Regression	OSR
Completeness	66%	94%
High Class Correctness	82%	81%
Average Correctness	76%	87%

Table 2: High Completion Cost Model Accuracies

The logistic regression results presented in the tables were obtained without using principal component analysis. Unexpectedly, the results were poorer when the principal components were used in the logistic regression equation, so we therefore decided to use the results obtained without the principal components.

In both result tables, the same phenomenon may be observed: logistic regression and OSR had similar results in terms of high class correctness, but OSR performed much better in terms of average correctness and completeness. The decision rules can be adjusted for either technique to allow, for example, a higher completeness (at the expense of correctness); however, in this example, the logistic regression technique can not achieve a level of completeness comparable to OSR without sacrificing correctness.

5 Lessons Learned Through Model Interpretation

In this section we will discuss the interpretability of logistic regression equations. Then we will interpret the generated OSR patterns in order to assess how they support our hypotheses about software reliability and modifiability. Through examples, we will demonstrate how OSR can be a useful tool in order to perform exploratory data analysis.

5.1 Interpretation of Logistic Regression Equations

As an experiment to assess the stability and therefore the meaning and interpretability of the calculated regression coefficient estimates, we recalculated the model several times the model calculated for completion effort. Each of the model's explanatory variables was successively removed from the equation and the model was recalculated. Table 5.3 show the variations of coefficient estimates. Each column is labelled with the removed explanatory variable. At a first glance, many explanatory variables become non-significant at the 0.05 level (flagged with *). Also parameters like LUUIOBJ, LUISUBP, LUEXC have a large variation in their associated coefficients, although they remain significant. Some of these phenomena are easily explained by looking at the correlation matrix of those variables. Strong direct correlations can be observed among several pairs of variables: R(LUUIOBJ, LUIOBJ)=0.816, R(LUISUBP,LUIOBJ)=0.543, R(LUISUBP, LUEXC)=0.447. However, these correlations cannot explain most the variation observable in Table 5.3, e.g., when LUIOBJ is removed, LUFNEMS becomes non-significant.

This instability may be explained by the unavoidable violation in many real world data sets of many of the important assumptions underlying regression analysis. Homoscedasticity is assumed but not guaranteed: although explanatory variables may be good predictors on a part of their range and non-significant elsewhere, regression assumes a predictor to be globally significant or not significant. Also, the significance of explanatory variables as predictors is strongly dependent of the context which is defined by the actual value of the other explanatory variables, e.g. the ratio of cascaded imports may be significant uniquely in the context where the number of imported parameters and subprograms is large. The straightforward question which can be asked

	None	LUEXC	LUPUDS	LUXTYP	LUFNEMS	LUIOBJ	LUISUBP	LUUIOBJ
intercept	3.990	1.444	3.160	2.658	1.280	1.215	2.180	1.600
LUEXC	-1.383	•	-0.870	-1.160	-0.760	-0.421	-1.040	-0.559
LUPUDS	0.086	0.023 *	•	0.067	0.060	0.000 *	0.004*	0.022*
LUXTYP	-0.238	-0.196	-0.195	•	-0.163	-0.107 *	-0.103 *	-0.148
LUFNEMS	-2.835	-0.608 *	-2.120	-1.997	•	-0.872 *	-1.917	-1.32
LUIOBJ	2.496	1.560	1.800	2.029	1.779	•	0.775	0.453
LUISUBP	-0.370	-0.028	-0.025	-0.027	-0.029	-0.001*	•	-.009
LUUIOBJ	-2.202	-1.410	-1.600	-1.824	-1.626	0.113 *	-0.703	•

Table 3: Instability of Regression Coefficient Estimates

when looking at the latter results is: are these coefficients interpretable (i.e. can we determine which ones have the strongest impact on the risk of having an error difficult to complete)? The answer is that only the coefficients that remained reasonably stable can be interpreted with a reasonable certainty. With respect to the other coefficients, it may be concluded that they have some difficult to quantify influence in some undetermined context.

5.2 Interpretation of OSR Patterns

In this section, we discuss the patterns generated by OSR. Then, we compare the interpretability of the respective patterns and regression equations. Some of the statistically significant, reliable patterns indicating high risk generated by the OSR process are presented and discussed. There are two groups of patterns, those related to isolation cost and those related to completion cost. The format in which the patterns are presented is described below. Assume we want to represent the following patterns:

(1) (Predicate$_k$ OR Predicate$_l$) AND Predicate$_m$

(2) (Predicate$_k$ OR Predicate$_l$) AND Predicate$_n$.

In this case, the patterns and associated information would be provided in the following format where Predicate$_m$ and Predicate$_n$ are defined in the context of Predicate$_k$ OR Predicate$_l$:

Predicate$_k$ OR Predicate$_l$
Statistics$_{kl}$
    Predicate$_m$:
    Statistics$_m$
    Predicate$_n$:
    Statistics$_n$

where Statistics is a set of the following fields:

- Variation in Entropy (ΔH) of the pattern : this represents the impact of a predicate in a determined context.
- Probability of being in the high risk class (PH)
- Number of Pattern Vectors (#PV) in the learning set matching the predicate in its context.

Predicates are of the form $EV_x \in SET_{xy}$, where EV_x is an explanatory variable, and SET_{xy} a subset of the value domain of EV_x.

5.2.1 Isolation Patterns

For the risk of having an error that is difficult to isolate, five major influences were found. These are: the number of imported declarations to the library unit aggregation (LUA), the size of the LUA, the degree of visibility control in the LUA, the locality of imports to the LUA, the extent of control flow in the LUA, and the number of user declared exceptions in the LUA. These influences are described in the following paragraphs, and will be discussed in the context in which they were determined to be significant. For each of these influential factors, examples of patterns associated with the factors are presented.

(1) number of imports:

LUISUBP ∈ [69%,100%] OR LUIPAR ∈ [75%,100%]
ΔH = 0.32, PH=0.82,#PV=39

LUIUDEC ∈ [72%,100%] OR LUITOT ∈ [72%,100%]
ΔH = 0.36, PH=0.84,#PV=37

LUCALLS ∈ [66%, 100%]
ΔH = 0.30, PH=0.81,#PV=42
    LUISUBP ∈ [35%, 100%]
    ΔH = 0.62, PH=0.926,#PV=27

A large number of imports to the LUA appears to be a significant indicator that the LUA may have a difficult to isolate error. There may be several reasons for this, since a large number of direct imports is often the result of

two influences: a large number of imported services, and a large number of compilation units that import the same service. On the other hand, a low number of imports appears to reduce the risk of having an error difficult to isolate. When there is little interaction with other library units, it may be easier for the programmer to isolate the origin of the error and to understand its consequences on the system functionalities. This phenomenon appears to be very influential according to the generated patterns since the corresponding predicates create in average the largest total variation of reliability. As indicated by the above patterns, this indication may be obtained early in development, e.g. by examining the number of imported subprograms (LUISUBP) or parameters (LUIPAR), or late, e.g. by examining the total number of imported declarations, LUITOT, or unique declarations (LUIUDEC, a similar count of imported declarations, but with overloaded declarations only counted once).

(2) Size of library unit aggregation:

LUSLOC ∈ [53%, 100%]
ΔH = 0.18, PH=0.74,#PV=58

LUOBJ ∈ [71%,100%] OR LUSLOC ∈ [71%,100%]
OR LUADA ∈ [70%,100%]
ΔH = 0.24, PH=0.78,#PV=41

LUCALLS ∈ [66%, 100%]
ΔH = 0.30, PH=0.81,#PV=42
LUOBJ ∈ [47%, 100%]
ΔH = 0.73, PH=0.95,#PV=22

The size of the LUA in question appears to be a significant indicator of the presence of a difficult to isolate error. When the LUA has a very small size, i.e. first quartile, errors are not as likely to appear, and when they do appear, they are not likely to be difficult to understand and isolate. On the other hand, the larger LUAs are much more likely to contain a difficult to isolate error. More information has to be analyzed in order to understand the structure and content of the LUA, adding to isolation effort. Several metrics are seen as such an indicator of a high risk component - from counts of object declarations (LUOBJ) to counts of statements (LUADA) and source lines of code (LUSLOC) in the component.

(3) Visibility Control:

LUVCPUD∈ [70%, 100%]
ΔH = 0.18, PH=0.74,#PV=35

The ratio of cascaded imports to direct imports [AES90] provides a crude measure as to whether declarations are being imported directly to where they are needed. A large ratio of cascaded imports to direct imports indicates that declarations are being imported into top level units in the library unit aggregation (e.g the LUA itself), and not into the low level units, where it is likely that the imported services are to be used. In this situation, to understand the interface of any single compilation unit, one must examine the interface of its ancestor units (from where the declarations were cascaded). This may result in additional error isolation effort.

(4) Control Flow:

LUAVECF∈ [63%, 100%]
ΔH = 0.17, PH=0.74,#PV=46

LUCALLS ∈ [66%, 100%]
ΔH = 0.30, PH=0.81,#PV=42

LUIEPUD ∈ [63%, 100%]
ΔH = 0.14, PH=0.86,#PV=46
LUCALLS ∈ [45%, 100%]
ΔH = 0.37, PH=0.91,#PV=11

Components with an excessive number of call branches are likely to be more difficult to understand and isolate an error, because of the additional paths that must be explored. LUAVECF, or the average number of call statements per subprogram in the LUA, was found to be an indicator of high isolation difficulty when it was in the uppermost quartile, supporting the hypothesis. LUCALLS, the count of call statements in the aggregation, is related to both control flow and size of the LUA. When this is large, there is a high probability that there will be a difficult to isolate error, supporting the hypotheses about size and control flow. Also, we see that LUCALLS provides an even stronger prediction when in the context of a large context coupling ratio (LUIEPUD), as is evidenced by the increased probability of being in the high risk class.

(5) Context Coupling ratio:

LUIEPUD ∈ [63%, 100%]
ΔH = 0.14, PH=0.72,#PV=46

LUIEUDEC∈[42%,100%] OR LUIETOT∈[42%,100%]
ΔH = 0.07, PH=0.66,#PV=73

One measure of design complexity suggested in [AE92] is context coupling, which measures the interconnection of compilation units. The ratio of imported to exported declarations was suggested as useful indicator of this type of complexity, as it accounts for the number of declarations made visible by context coupling, normalized by the number of exports in the library unit. We see that as this ratio increases, the likelihood of a difficult to isolate error also increases. Again, we see this influence both with measures available early (with the ratio measured by program unit declarations,

LUIEPUD), and late, measured by total declarations and unique declarations (LUIETOT and LUIEUDEC).

(6) Number of exceptions:

LUEXC ∈ [76%, 100%]
ΔH = 0.28, PH=0.80,#PV=30

One interesting frequently occuring pattern indicating a high risk component included the number of user declared exceptions (LUEXC) being in the uppermost quartile. Exception handling is an often overlooked and misunderstood feature of the Ada language; this pattern indicates that there may have been difficulty with it in this environment. Further investigation would be necessary to confirm this, but, in any event, it does serve as a useful indicator of a high risk component.

(7) Locality of imports:

LUIOUDEC ∈ [84%, 100%]
ΔH = 0.26, PH=0.21,#PV=19

We expected that having most imports originate locally would reduce the likelihood of such a high risk error, as the designer(s)/programmer(s) may have a greater familiarity with artifacts of his own subsystem than with those of other subsystems. LUIOUDEC is a measure of the fraction of imported unique declarations that are declared in the same subsystem as the LUA in question. We see that when it is extreme, i.e. most to all imports come from "local" units, there is a low probability (0.21) of being in the high risk class.

5.2.2 Completion Patterns

Here again, several phenomena related to the assumptions made in section 3 may be observed from these patterns:

(1) Visibility Control:

LUVCPUD ∈ [58%, 100%]
ΔH = 0.13, PH=0.71,#PV=62

LUSLOC ∈ [68%, 100%]
ΔH = 0.10, PH=0.68,#PV=47
LUVCTOT ∈ [25% ,100%]
ΔH = 0.34, PH=0.83,#PV=35

LUUITOT ∈ [69%, 100%]
ΔH = 0.11, PH=0.69,#PV=46
LUVCTOT ∈ [36% ,100%]
OR LUVCUDEC∈ [43% ,100%]
ΔH = 0.42, PH=0.86,#PV=29

LUINST ∈ [76%, 100%]
ΔH = 0.22, PH=0.77,#PV=35
LUVCUDEC ∈ [45% ,100%]
OR LUVCTOT ∈ [45% ,100%]
ΔH = 0.70, PH=0.95,#PV=19

A large ratio of cascaded imports to direct imports raises the risk of having an associated difficult to complete error. This was typically found to be significant in the context of a large LUA or a LUA that contains a large number of imports (cascaded or direct). If declarations are not imported directly to where they are needed, it may result in additional effort to understand the unit, which may result in additional error correction effort.

(2) Number of imports:

LUIUDEC ∈ [77%, 100%]
ΔH = 0.09, PH=0.67,#PV=49
LUCUDEC ∈ [48% ,100%]
ΔH = 0.37, PH=0.84,#PV=25

LUCUDEC ∈[60%,100%] OR LUCTOT ∈[58%,100%]
ΔH = 0.09, PH=0.68,#PV=62

A large number of imports (i.e. subprograms, types, subtypes, formal parameters) to a library unit aggregation appears to increase the risk of having an error difficult to complete a change. This appears whether the imports are counted in terms of direct imports or cascaded imports. As explained previously, while this may be a due to a library unit aggregation requiring the services of an excessive number of other LUAs, it may also be an indicator of the size of the aggregation itself; since multiple compilation units in the larger LUAs often import the services of the same LUA, thereby increasing the measures found in the above predicates. When there is less interaction with other LUAs, it may be easier to implement the change and evaluate its consequences on the system functionalities. As with the effort to isolate a change, this phenomenon appears to be very frequent and influential. The influence can be seen as measured by direct (LUIUDEC) and cascaded (LUCTOT, LUCUDEC) imports.

(4) Size of LUAs:

LUPUDS ∈ [65%, 100%] OR LUSUBP ∈ [65%, 100%]
ΔH = 0.09, PH=0.67,#PV=52

LUOBJ ∈ [52%, 100%]
ΔH = 0.05, PH=0.63,#PV=72

If the LUA has a very large size, then errors are more likely to appear and changes are more likely to be difficult to complete. It is expected to see large LUAs to

be likely to require additional effort to understand, correct, and verify. Here, we see patterns that indicate that LUAs containing many program unit declarations (LUPUD), subprogram declarations (LUSUBP), and object declarations (LUOBJ) are more likely to be in the high cost class.
(5) Number of exceptions:

LUEXC ∈ [73%, 100%]
ΔH = 0.19, PH=0.75,#PV=40

As with the isolation cost models, when there are many exceptions declared there is an increased probability of a difficult to complete error. These patterns may be indicative of problems in controlling exception handling.

(6) Count of Instantiations:

LUINST ∈ [76%, 100%]
ΔH = 0.22, PH=0.77,#PV=35

LUCUDEC ∈ [60%,100%] OR LUCTOT ∈ [58%,100%]
ΔH = 0.09, PH=0.68,#PV=62
LUINST∈ [56% ,100%]
ΔH = 0.50, PH=0.89,#PV=27

LUAs with a relatively large number of instantianted generics (LUINST) were found to be likely to have a difficult to complete error. This is even more significant in the context of a large number of cascaded imports. As with the previously noted difficulty with exceptions, this may be an indicator of difficulty with the Ada generic features. Again, further investigation would be necessary determine this.

(7) Parameterization:

LUAVECF ∈ [0%, 57%] OR LUCALLS ∈ [0%, 56%]
ΔH = 0.04, PH=0.38,#PV=92
LUPARM∈ [66%,100%] OR LUPARPV ∈ [72%, 100%] OR LUPARPD ∈ [72%, 100%]
ΔH = 0.30, PH=0.19,#PV=32

This pattern focuses on the parameterization of the imported units. When we have a well parameterized unit, i.e. a large number declared parameters (LUPARM), or a high ratio of parameters per program unit declaration (LUPARPD), or visible parameters per visible program unit declaration (LUPARPV), we see a low probability (0.19) of a difficult to complete error.

5.2.3 Discussion of Pattern Interpretability

As we have seen in the above examples, in contrast to the regression equations, patterns are more suitable for interpretation for the following reasons:

(1) They explicitly describe the context in which predicates appear to be significant predictors.

(2) The impact of a predicate is only dependent on the defined context as opposed to regression parameters that may be sensitive to many parameters in the regression model. This indicates that the patterns will be stable, which our generated regression models were not.

(3) They show explicitly the associations in various contexts between exploratory variables.

(4) They explicitly define the range on which a variable appears to be an accurate predictor.

6 Conclusions

We can draw three major conclusions from these experimental results:

(1) With respect to Ada systems, it seems possible to build accurate risk models during the design phase to help designer prevent difficulties and testers manage their resources. In other words, we have shown that it may be possible to construct models which facilitate cost benefit analysis using model correctness and completeness. The analysis may be used to make decisions concerning remedial actions during development.

(2) The Optimized Set Reduction approach seems to be a good alternative for multivariate empirical modeling in this application domain since the pattern-based classification appear more accurate than those from logistic regression equations. This also confirms previous studies showing similar results for other kinds of applications [BBT92, BBH92].

(3) Patterns appear to be more stable and more interpretable structures than regression equations when the theoretical underlying assumptions are not met. This is a very important point in the context of the improvement paradigm [BR88]. Feedback and therefore process improvement is only possible if the generated quantitative models are interpretable. Taking effective corrective actions is only possible when the impact of controllable factors on the parameters to be controlled (e.g. cost or quality) can be fully understood and quantified.

The primary limitations of the OSR approach are the following:

(1) OSR being a search algorithm, computation is more intensive than for an analytical model.

(2) OSR may be comparatively less accurate when the assumptions underlying the logistic regression analysis are met.

7 Acknowledgments

We would like to thank Victor Basili for his comments on this paper. Also, we would like to thank William Agresti, Frank McGarry and Jon Valett for their support in providing the data used in this analysis.

8 References

[Agr90] A. Agresti, *Categorical Data Analysis*, John Wiley & Sons, 1990.

[AES90] W. Agresti, W. Evanco, and M. Smith, "Early Experiences Building a Software Quality Prediction Model", *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November, 1990.

[AE92] W. Agresti and W. Evanco, "Projecting Software Defects from, Analyzing Ada Designs", *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[AE+92] W. Agresti, W. Evanco, D. Murphy, W. Thomas, and B. Ulery, "Statistical Models for Ada Design Quality", *Proceedings of the Fourth Software Quality Workshop*, Alexandria Bay, New York, August, 1992.

[Bas85] V. Basili, "Quantitative Evaluation of Software Methodology", *Proceedings of the First Pan Pacific Computer Conference*, Australia, July 1985.

[BR88] V. Basili and H. Rombach,"The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

[BF+84] L. Breiman, J. Friedman, R. Olshen and C. Stone, *Classification and Regression Trees*, Wadsworth & Brooks/Cole, Monterey, California, 1984.

[BP92] 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, April 1992.

[BBH92] L. Briand, V. Basili and C. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development", *IEEE International Symposium on Software Reliability Engineering*, North Carolina, October 1992.

[BBT92] L. Briand, V. Basili and W. Thomas, "A Pattern Recognition Approach for Software Enginnering Data Analysis", *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[CA88] D. Card and W. Agresti, "Measuring Software Design Complexity", *Journal of Systems and Software*, 8 (3), March, 1988.

[DG84] W. Dillon and M. Goldstein, *Multivariate Analysis: Methods and Applications*, Wiley and Sons, 1984.

[Dou87] D. Doubleday, "ASAP: An Ada Static Source Code Analyzer Program", TR-1895, Department of Computer Science, University of Maryland, August, 1987.

[EA92] W. Evanco and W. Agresti, "Statistical Representations and Analyses of Software", *Proceedings of the 24th Symposium on the Interface of Computing Science and Statistics*", College Station, Texas, March, 1992.

[GKB86] J. Gannon, E. Katz, and V. Basili, "Metrics for Ada Packages: An Initial Study", *Communications of the ACM*, 29 (7), July, 1986.

[HK81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Trans. Software Eng.*, 7 (5), September, 1981.

[MK92] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs", *IEEE Trans. Software Eng.*, 18 (5), May, 1992.

[PA+82] H. Potier, J. Albin, R. Ferreol and A. Bilodeau, "Experiments with Computer Software Complexity and Reliability", *Proceedings of the Sixth International Conference on Software Engineering* , September, 1982.

[Qui86] J. Quinlan, "Induction of Decision Trees", *Machine Learning* 1, Number 1, 1986.

[Rom87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Trans. Software Eng.*, 13 (3), March, 1987.

[SP88] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis", *IEEE Trans. Software Eng.*, 14 (12), December, 1988.

An Information Model for Use in Software Management Estimation and Prediction

Ningda R. Li and Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

This paper describes the use of cluster analysis for determining the information model within collected software engineering development data at the NASA/GSFC Software Engineering Laboratory. We describe the Software Management Environment tool that allows managers to predict development attributes during early phases of a software project and the modifications we propose to allow it to develop dynamic models for better prediction of these attributes.

Keywords: Cluster analysis; Data modeling; Measurement; Software management; Tools

1 Introduction

Software management depends upon managers to collect accurate data of the software development process and on the production of accurate models upon which to use that data. Lines of code is still the most widely used measure for cost and error analysis, even though it is known to be inaccurate [8]. However, since it is not known until the completion of a project, its use as a predictive measure is not reliable. What are needed are more accurate models of the software development process.

Current models are developed according to broad categories, such as waterfall development, spiral model development, cleanroom development, etc., with additional qualifiers giving a few attributes of the product (e.g., real time, embedded application, data base). Data is often collected and projects are compared to historical baselines according to these general categories. For example, the COCOMO model [1] is based upon a small set of predefined factors, and predictions are made according to how a new project measures up to these factors.

It is difficult for software managers, however experienced they are, to evaluate the status or quality of a software development project and make correct decisions without accurate, reliable measurement models and data. These data include metrics aimed at clarifying and quantifying some quality of either a software product, or the development process itself [13].

Since we do not have accurate models of the software development process, perhaps, we can use the data itself to develop dynamic models of software development that reflect the changing nature of the development process. In this paper we study one particular modeling technique, cluster analysis, as a means for determining the underlying information model present in the collected software engineering development data.

The importance of software management has led to the development of various software management tools for aiding in this effort. These tools help software managers get access to, visualize, and analyze measurement data. The Software Management Environment (SME) is one of those tools developed within the NASA Goddard Space Flight Center Software Engineering Laboratory (SEL) [6], [12]. It is the purpose of this paper to investigate the use of cluster analysis within SME to enhance the ability of software managers to predict and control the software development process.

In Section 2 we describe the information model and the measures used by SME. In Section 3 we describe

Second International Conference on
Information and Knowledge Management
Arlington, VA
November, 1993

our use of cluster analysis to dynamically change our information model, and in Section 4 we describe some preliminary results of using our new model. We then give our conclusions to this work.

2 Measurement in SME

For over 15 years the software engineering community has been studying various models of the software development process. Concepts like Halstead's software science measures, Putnam's Rayleigh curve, Boehm's COCOMO model, among many others, are all attempts at providing a quantitative model underlying the software development cycle. Unfortunately, most of these models are very general, and while broadly describing the software process, do not have the granularity to make accurate predictions on a single software project.

As a way to further these studies, the Software Engineering Laboratory was established to evaluate the above models and develop new models within a production programming environment.

2.1 NASA/GSFC SEL

The NASA Goddard Space Flight Center Software Engineering Laboratory is a joint research project of GSFC Flight Dynamics Division, Computer Sciences Corporation and the University of Maryland. Data from over 100 projects has been collected since 1976 and a data base of over 50 Mbytes of measurement data has been developed. Initially supporting 100,000 line FORTRAN ground support software for unmanned spacecraft written by 10 to 15 programmers over a 2 year period for an IBM mainframe, the SEL data base now includes a wider variety of projects consisting also of Ada and C code for a variety of machines.

The SEL collects data both manually and automatically. Manual data includes effort data (e.g., time spent by programmers on a variety of tasks - design, coding, testing), error data (e.g., errors or changes, and the effort to find, design and make those changes), and subjective and objective facts about projects (e.g., start and end completion dates, goals and attributes of project and whether they were met). Automatically collected data includes computer use, program static analysis, and source line and module counts.

2.2 Measure Models

Data modeling often combines various measures in order to evaluate attributes in a software development. For example, classification trees were used as part of the Amadeus project [9][10] and a variant of that method was used within the SEL [11]. In this case, a tree is generated where each leaf node represents one of several results. Based upon values for each measure, a path down the tree is taken until a result at a leaf node is reached.

For each project, we can compare the collected data over time with a predefined model of a similar project from the data base. A *basic measure model* refers to the expected behavior of a software development measure as a function of time [5]. Measures, developed from the raw data collected by the SEL, include lines of code, staff hours, computer hours, and changes and errors. A measure model is usually obtained by examining the data for that measure over a set of projects and averaging them. Time is described in terms of the four major phases of software development within the waterfall life-cycle: design, code and unit test, system test, and acceptance test.[1] Measure behavior is described in terms of percent completion of that measure at each distinct checkpoint.

Within the SEL, we describe one of these measure models as a vector of 15 points, each representing the percent completion of the measure at distinct dates in the development cycle (generally 25% increments through each phase). Table 1 shows the tabular representation of a Lines of Code (LOC) model [5] and Figure 1 shows the graphical representation of the same model. According to the LOC model, no code should be written during the design phase, and most of the code (76%) should be written during the code and unit test phase.

For ease of use, we can use the vector representation of the model:

$$[0, 0, 0, 0, 0, 6.86, 36.05, 53.99, 76.28, 86.82, 94.88, 96.09, 98.14, 99.58, 100]$$

In general, a measure model can be represented by the following vector:

$$P = [p_0, p_1, p_2, \ldots, p_{13}, p_{14}]$$

[1] The SEL does not collect specification data since that task is performed by another group. This is reflected in the models that the SEL develops, and is a good indication why no two development models are easily transportable across locations.

Phase	% of Phase	% of Total Lines
Design	0	0.00
	25	0.00
	50	0.00
	75	0.00
Code/Unit	0	0.00
Test	25	6.86
	50	36.05
	75	53.99
System Test	0	76.28
	50	86.82
Acceptance	0	94.88
Test	25	96.09
	50	98.14
	75	99.58
End	100	100.00

Table 1: Tabular representation of a LOC model

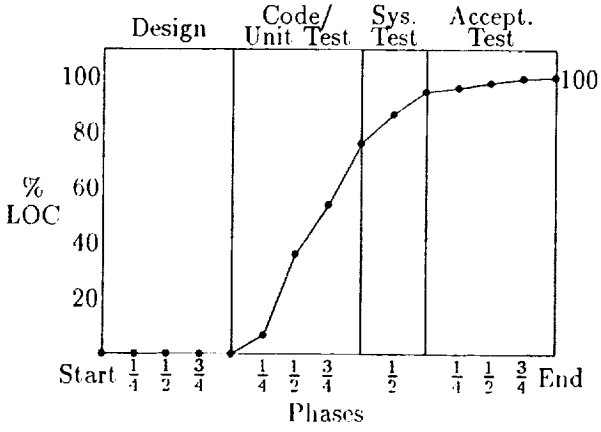


Figure 1: Graphical representation of a LOC model

with $p_0 = 0$, $0 \leq p_i \leq 100$ for $1 \leq i \leq 13$, and $p_{14} = 100$.

We will use *measure pattern* to refer to a measure model derived from a single project. Essentially, we produce a measure model as the average of some set of measure patterns.

2.3 SME

The Software Management Environment was developed to help software managers carry out management activities like observation, comparison, prediction, analysis, and assessment [6]. In order to provide these functions, SME uses measurement data from current and past projects from the SEL database, research results in terms of models and relationships, and manager experience from the past.

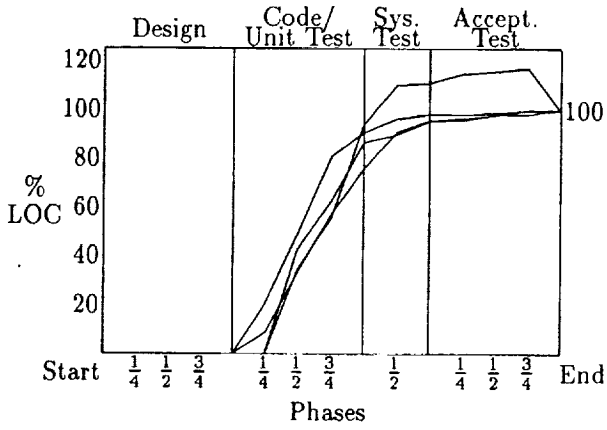


Figure 2: LOC patterns

SME was initially built with a fixed set of measure models. For example, for LOC (lines of code), the most apparent predictor seemed to be programming language. Therefore, SME originally had two models of LOC based upon language - Ada and FORTRAN. Each project was classified according to the measure model it was expected to adhere to, and for each measure type, a predefined measure model was stored in the data base.

Some of the features of SME are described below.

Measure models in SME

Currently in SME, a measure model is derived from a set of projects with the same characteristics, such as development methodology, programming language, and development environment. SME decides which measure model to use for a project measure of interest based on the characteristics of that project. For example, Figure 2 shows four LOC patterns of four different projects with the same characteristics. SME creates a LOC model by averaging these patterns, but is the resulting model a good representative of actual LOC behavior? This is the basic question behind our research plan, and our goal is to develop, dynamically, LOC (and other) models that better represent attribute behavior.

Observation and Comparison

To monitor the progress of a project, managers need

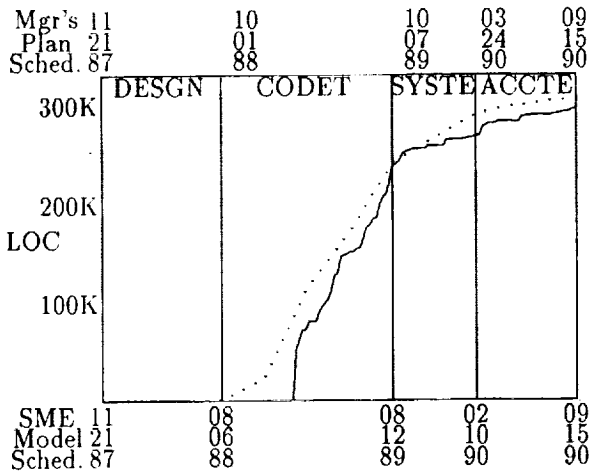


Figure 3: Growth in 'Lines of Code' for P_1

cumulative growth data for measures such as effort, size and errors. SME provides graphic display of the actual collected data like shown in Figure 3, in which the solid curve represents an overall view of project P_1's growth in size (lines of code) over a specified calendar time. The dotted curve in Figure 3 shows a LOC model of a similar project or the LOC measure model from the data base to permit the manager to compare project data to a model which indicates the "normal behavior" for such projects. Comparison can also be made between projects.

Prediction

SME can also predict a measure's completion value for an on-going project, by using the appropriate measure model scaled up to the actual time schedule of the new project. Using the initial data collected from a project, final values can be estimated giving the manager an indication of the measure's possible future behavior.

Analysis and Assessment

SME can help the manager identify the probable causes of any unexpected behavior for a measure, and assess the quality of a project based on all the measurement data. For each measure, a knowledge base of cause-effect relationships is maintained. So, for example, if a given project seems to have too many errors at a certain point in the coding phase compared to the error measure model, a rationale can be provided to the manager, such as:

```
TEAM IS REPORTING INCONSEQUENTAIL ERRORS
INEXPERIENCED DEVELOPMENT TEAM
POOR USE OF METHODOLOGY
COMPLEX PROBLEM
etc.
```

Similar idea can be found in [4]. What is desired is a mechanism whereas this knowledge base can be updated dynamically as projects evolve.

3 Cluster analysis

Cluster analysis is the technique for finding groups in data [7] that represent the same information model. Biologists and social scientists have long used it to analyze their data. Here, we use it to find similar measure patterns within the collected software development data.

Clustering was used previously in an early SEL study [3] in order to determine possible patterns in projects by clustering the modules that make up the project. The results were somewhat inconclusive due to large variances within small modules and the many different attributes that contributed to the single value that was clustered. In this current study, we try to separate out different attributes and study their effects over time. This gives greater precision to the data we are looking at and eliminates much of the variability found in the earlier study.

3.1 Clustering

As stated in section 2.2, a measure pattern can be represented by a vector. Clustering is a method to determine which vectors are similar and represent the same or similar physical objects. There are several clustering and modeling algorithms, including:

- *Euclidean distance.* Each vector represents a point in n-space. Points near one another are in the same cluster.
- *Cosine.* Each measure pattern represents a vector from the origin. The cosine of the angle between two vectors represents the similarly in their components and hence their closeness.
- *Optimal Set Reduction.* OSR generates, based on search algorithms and univariate statistics, logical expressions which represent strong patterns in a data set [2].

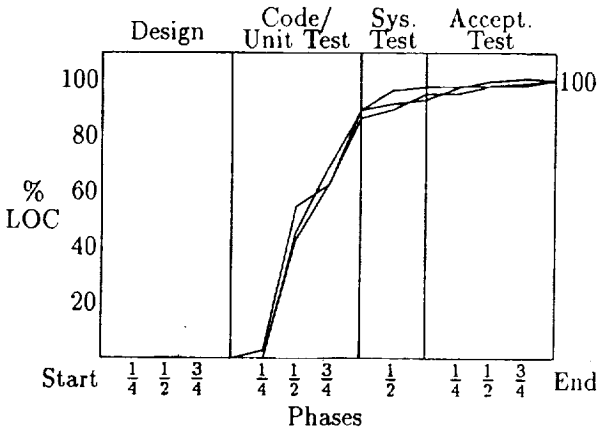


Figure 4: A cluster of LOC patterns

Several other algorithms also have been used.

For our initial investigation, we are using the Euclidean distance between two vectors as a degree of similarity between two measure patterns. For example, if $P = [p_0, p_1, p_2, \ldots, p_{13}, p_{14}]$ and $N = [n_0, n_1, n_2, \ldots, n_{13}, n_{14}]$ are two measure patterns, then their *Euclidean distance* is

$$ed(P, N) = \sqrt{(p_0 - n_0)^2 + \cdots + (p_{14} - n_{14})^2}$$

Two patterns are assumed similar and are in the same cluster if and only if $ed(P, N) < \varepsilon$.

Note that by varying ε we can adjust the size of the clusters by specifying how close two vectors must be in order to be in the same grouping. Since single vector clusters provide no information, we want to adjust ε so that we generally have clusters of at least 3 vectors without including vectors that represent fundamentally different curves. Figure 4 shows a cluster of three LOC patterns.

3.2 Cluster model

A *cluster model* is the average of all measure patterns in one cluster. It closely describes the measure behavior for all projects in the cluster because measure patterns in the same cluster are similar. Instead of choosing a predefined measure model for a project measure of interest using the project's characteristics (as is currently the case with SME), a cluster model can be dynamically selected for the project measure depending on which cluster its pattern best fits.

A further advantage from the current static approach of SME, is that alternative models can be developed for each measured attribute. Within SME, the same measure model is used for all measured attributes. For example, if the defining characteristic is Ada for the LOC measure, it will be the Ada measure model for each other measure (e.g., error, effort). With dynamic clustering, measure models can vary for each distinct measure.

For an ongoing project, a manager's estimate of schedule and measure completion values are used to derive its measure patterns. Estimates are replaced by real data once they become available. So a project measure's closest cluster model may change as the project develops. In Section 4.3 we discuss how to use this information to improve on the predictive capabilities of SME. On the other hand, since a project's development methodology or programming language usually do not change during a project's development life-cycle, the static measure model chosen by the current implementation of SME based on those characteristics does not change.

Similarly, SME does an assessment of a project's real data compared to the measure model's estimate by use of a predefined set of attributes. But by looking at the attributes that are common for all projects within a given cluster, we may be able to determine general characteristics for any new project that falls within that cluster. This list of attributes will dynamically evolve over time instead of being a static description of project behavior. For example, if all projects within a given cluster were previously late in delivery, it may be useful to report this information to the manager of a new project that falls within this cluster.

This allows the knowledge base to grow and change dynamically as projects develop. It does not require the predefinition of a few models - which may not even accurately represent the actual development model, only a manager's poor estimate of one.

4 Evaluation of Clustering

Before implementation of our clustering approach within SME, we evaluated the effectiveness of clustering with a subset of the SEL data base. Measurement data from twenty-four projects in the data base were clustered using eight different measures: computer hours (CPU), total staff hours (EFF), lines of code (LOC), modules changed (MCH), module

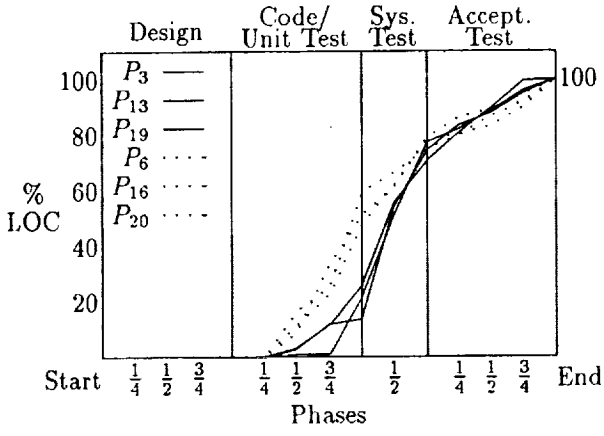


Figure 5: Two clusters of MCH patterns

count (MOD), reported changes (RCH), reported errors (RER), and computer jobs (RUN). We then studied common objective and subjective attributes of projects in the same cluster.

For example, Figure 5 shows two clusters of MCH (module changes) patterns. Cluster C_1 consists of patterns from projects P_3, P_{13} and P_{19}, and cluster C_2 consists of patterns from projects P_6, P_{16}, P_{20}. We observe that more than half of the module changes were made during the code and unit test phase for projects in C_2 compared to about twenty precent for projects in C_1. Consequently, only twenty percent of the module changes were made during the system test phase for C_2 compared to about fifty percent for C_1.

4.1 Objective characteristics

Project characteristics of the two clusters are summaried in Table 2 and 3 respectively. We observe that if computer language is the basis for choosing a MCH measure model, as is the case with the current version of SME, all six projects will use the same MCH model since they all use FORTRAN. In this case, clustering discovers the two vastly different behaviors of MCH measures which are undetectable with the static approach.

In addition, some commonly used discriminators do not appear to be significant with these clusters. Size is often used to classify projects, yet cluster C_1 contains projects from 16K to 179K source lines. The projects represent two very different hardware and software environments (IBM mainframe and DEC VAX VMS) and each project in C_1 represents a different application area. (However projects in C_2 are more homogeneous; they all represent relatively large 168K to 296K attitude ground support systems built as mainframe IBM applications.)

Attributes	P_3	P_{13}	P_{19}
Computer	IBM	DEC	IBM
Language	FORT.	FORT.	FORT.
Application	AGSS	SIM.	ORBIT
Reuse (%)	10.1	30.7	38.1
Time (wks)	116	119	109
Size (SLOC)	178.6	36.6	15.5

Table 2: Project characteristics for cluster C_1

Attributes	P_6	P_{16}	P_{20}
Computer	IBM	IBM	IBM
Language	FORT.	FORT.	FORT.
Application	AGSS	AGSS	AGSS
Reuse (%)	19.5	1.9	10.0
Time (wks)	97	87	147
Size (SLOC)	167.8	233.8	295.4

Table 3: Project characteristics for cluster C_2

4.2 Subjective characteristics

Subjective data for each project is stored in the data base as an integer between 1 (low or poor) and 5 (high). Each project manager fills in these values at the end of a project based upon experiences during the development. For each cluster we retrieved those subjective attributes that differed by at most 1 within the cluster, thus indicating a common feature for those clustered projects. This information can then be fed back to the manager of a new project that falls within that cluster to provide an indication of probable future behavior.

Projects in cluster C_2 have common ratings on the following subjective attributes:

```
Tightness of schedule constraints: 3
Access to development system: 3
Timely software delivery: 4
```

We notice that their rating for timeliness of software delivery is relatively high. This could be a direct result of the fact that most module changes were made during code and unit test phase.

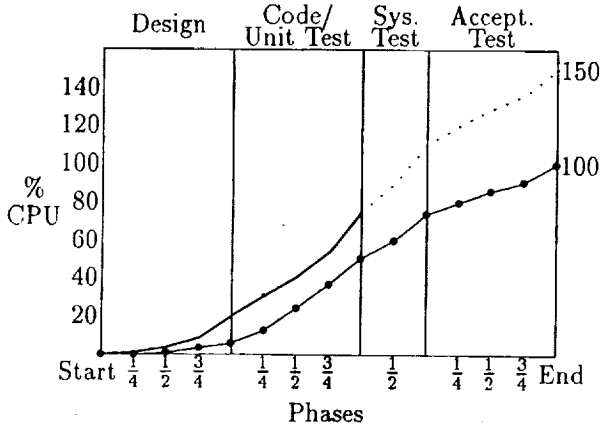


Figure 6: Prediction for CPU

4.3 Predictive models

The two clusters of Figure 5 are easiest to measure when all data points for each measure model are available. However, it is the very nature of predictive models that some of this data is incomplete. We are currently altering SME's predictive capabilities to take this into account.

If data is available for new project P up through point i (e.g., values for $p_0, p_1, \ldots, p_i$), then clustering for P against each existing cluster will be only with respect to these $i+1$ points. That is, for each cluster C, it will be assumed that p_i and c_i have the same value and P's other values will be scaled accordingly. Clustering will determine which cluster has the closest shape to P's shape.

Once a matching cluster is found, it will be assumed that project P has the same characteristics as this found cluster and the succeeding values for P will match the cluster's measure model for points $i+1$ through 14.

The effect will be to scale P's original estimate with respect to the cluster's estimate. For example, in Figure 6, if the cluster estimated a 50% completion by point 8 and the actual data showed a 75% "completion," then it can be assumed that the actual completion will be 150% since the relevant cluster is only half finished. In this case it can be assumed that the manager underestimated the resources needed for this project. We are currently modifying SME's graphical interface to show these predicted curves.

The predictive model for project P depends upon both estimating the total resources needed in order to compute the percentage for point P_i and estimating the schedule in order to determine how far one has progressed in the current development phase. Either one, however, may not be accurate. For example, current point P_6 represents 50% coding, yet that is only known when coding is complete. The current date may possibly range from perhaps the 25% level (and hence really represent point P_5) to the 75% level (and hence really represent point P_7) depending upon how accurately the initial schedule was set up. The true date will be known only after the coding phase is completed. However, in the above paragraphs we have described a mechanism to estimate resource needs when we assume that the schedule is correct.

On the other hand, if the latest available project point P_i is scaled to a cluster model horizontally along phases instead of vertically (i.e., by changing the estimated schedule), we can predict future changes in project schedule. However, since only discrete milestones of a schedule are used, they need to be quantified before numerical scaling can be applied. We are looking at extending the SME predictive model in order to estimate both the resource needs as well as potential bounds on the schedule based upon current data.

It should be realized that the model's predictive capabilities improve as a project develops. Very few points are available for prediction early in the development cycle leading to few differences among the various clusters. On the other hand, late in the development cycle where there is more variability among the clusters, it may be too late to change development models to account for any potential problems. How well the early predictions lead to significant differences in project development attributes is obviously an issue we need to investigate.

4.4 Model evaluation

Aside from its primary use as a tool to aid management in predicting future behavior on a current software development project, use of cluster analysis permits SME to be used as a tool to evaluate new models. If a model is proposed that describes some attribute of development that is collected by the SEL data base, then all projects within a cluster should exhibit that attribute to a great extent.

For example, the SEL is currently planning to en-

hance the SEL data base with additional predefined measure models in addition to the two models used at present. Often the following attributes (and their relevant values at NASA) are viewed as important attributes of a development methodology:

- *Computer use* – IBM or DEC environment
- *Reuse of existing source code* – Low, medium or high reuse of existing source code
- *Language* – FORTRAN or Ada as a source programming language
- *Methodology* – Cleanroom or standard NASA waterfall development method

By choosing one value from each category, the SEL can develop 24 possible models. A subset of these will be built into the SEL data base as predefined models for each project and each project will be assigned to one of these categories. However, while they are often viewed as crucial attributes, are these really discriminators useful to differentiate among projects?

If these are really discriminators of project development, then projects within a single cluster should all consist of the same predefined measure model (or at least predominately so). We can then use our clustering approach to determine the effectiveness of the new proposed models.

We can also use clustering to determine if there are any relationships among measures. If a cluster for Reported Change (RCH) consists of the same projects as a cluster for Reported Error (RER), this indicates that those two measures are closely related. If projects A and B are in the same cluster for CPU, LOC and RUN, then those projects are somewhat related.

This approach can be extended to any quantitative model. Projects in the data base can be grouped according to how well they meet the discriminators of any new proposed measure. The projects can be clustered, and if the models are appropriate, then clusters should be somewhat homogeneous.

For example, cleanroom is a technique that addresses early verification of a design that should result in fewer resulting errors (with less testing necessary) later in the development cycle. If so, then measuring reported errors (RER) per computer run (RUN) should cluster cleanroom projects together, and the plots should show high measure model values early in the development cycle. We can use SME to test such claims from this and other proposed models.

4.5 Evaluation of clustering

Clustering is effective in distinguishing measure behaviors. For most of the measures studied, we were able to yield clusters that differentiated behavior among the projects, whereas the current SME would consider them all similar and use the same measure model on that data.

A current weakness, however, is that the resulting clusters yield few common objective or subjective characteristics. We believe that this is due more to the nature of the current subjective files within the SEL data base than in the clustering methodology itself. The current data files are developed by the project managers and contain attributes about the project (e.g., external events such as schedule and requirements changes, team composition, environment composition). There is little about how management was performed (e.g., we didn't test enough, we started coding too soon). This is understandable given how the data was collected. We need to develop methods to collect this latter data in a non-threatening manner from each project manager so that it can be fed back to future project managers more effectively.

5 Conclusion

In this paper, clustering is presented as a mechanism for dynamically determining and altering the information model that describes certain attributes of the software development process. This permits the software manager to more accurately predict the future behavior of a given project based upon similar characteristics of existing projects in a data base. We believe the resulting cluster models are fairly accurate indicators of such behavior.

Clustering also permits rationale for deviations from normal behavior to be determined dynamically and are easier to generate than the existing expert system approach. Preliminary evaluation of clustering leads us to believe that the resulting models are fairly accurate indicators of such behavior.

In addition, it appears that some often used discriminators may not be totally effective in classifying projects. Size, programming environment and application domain may unnecessarily separate projects into categories that are ultimately the same (e.g., see

Tables 2 and 3). Obviously, this needs further study.

We are in the process of modifying NASA/GSFC's SME management tool for incorporation of these new models into the tool. We believe that this should greatly improve SME's predictive capabilities. Modification of the data in the SEL subjective data files should greatly aid in the analysis and assessment aspects of SME.

However, the process is far from over. We also intend to study alternative clustering and modeling techniques (e.g., Optimal Set Reduction, Cosine) in order to determine the best approach towards measuring these critical attributes. In addition, we need to observe how well early predictions of a project match with subsequent observations in order to be able to use SME as an effective management planning and tracking tool.

6 Acknowledgement

This research was supported in part by grant NSG-5123 from NASA/GSFC to the University of Maryland. We would like to acknowledge the contribution of Jon Valett of NASA/GSFC and Robert Hendrick of CSC as major developers of the original SME system and for their and Frank McGarry's (also of NASA) helpful advice on proposed changes we are making to SME.

References

[1] B. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[2] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Providing an empirical basis for optimizing the verification and testing phases of software development. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, Research Triangle Park, NC, October 1992.

[3] E. Chen and M. V. Zelkowitz. Use of cluster analysis to evaluate software engineering methodologies. In *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, March 1981.

[4] R. Chillarege, I. S. Bhandari, and et al. Orthogonal defect classification – a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[5] W. Decker, R. Hendrick, and J. Valett. The software engineering laboratory (sel) relationships, models, and management rules. Technical Report SEL-91-001, The Software Engineering Laboratory, NASA Goddard Space Flight Center, Greenbelt, MD, February 1991.

[6] R. Hendrick, D. Kistler, and J. Valett. Software management environment (sme) concepts and architecture (revision 1). Technical Report SEL-89-103, The Software Engineering Laboratory, NASA Goddard Space Flight Center, Greenbelt, MD, September 1992.

[7] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley & Sons, New York, NY, 1990.

[8] R. E. Park. Software size measurement: A framework for counting source statments. Technical Report 92-TR-20, Software Engineering Institute, Carnegie Mello University, Pittsburgh, PA, September 1992.

[9] A. A. Porter and R. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, 1990.

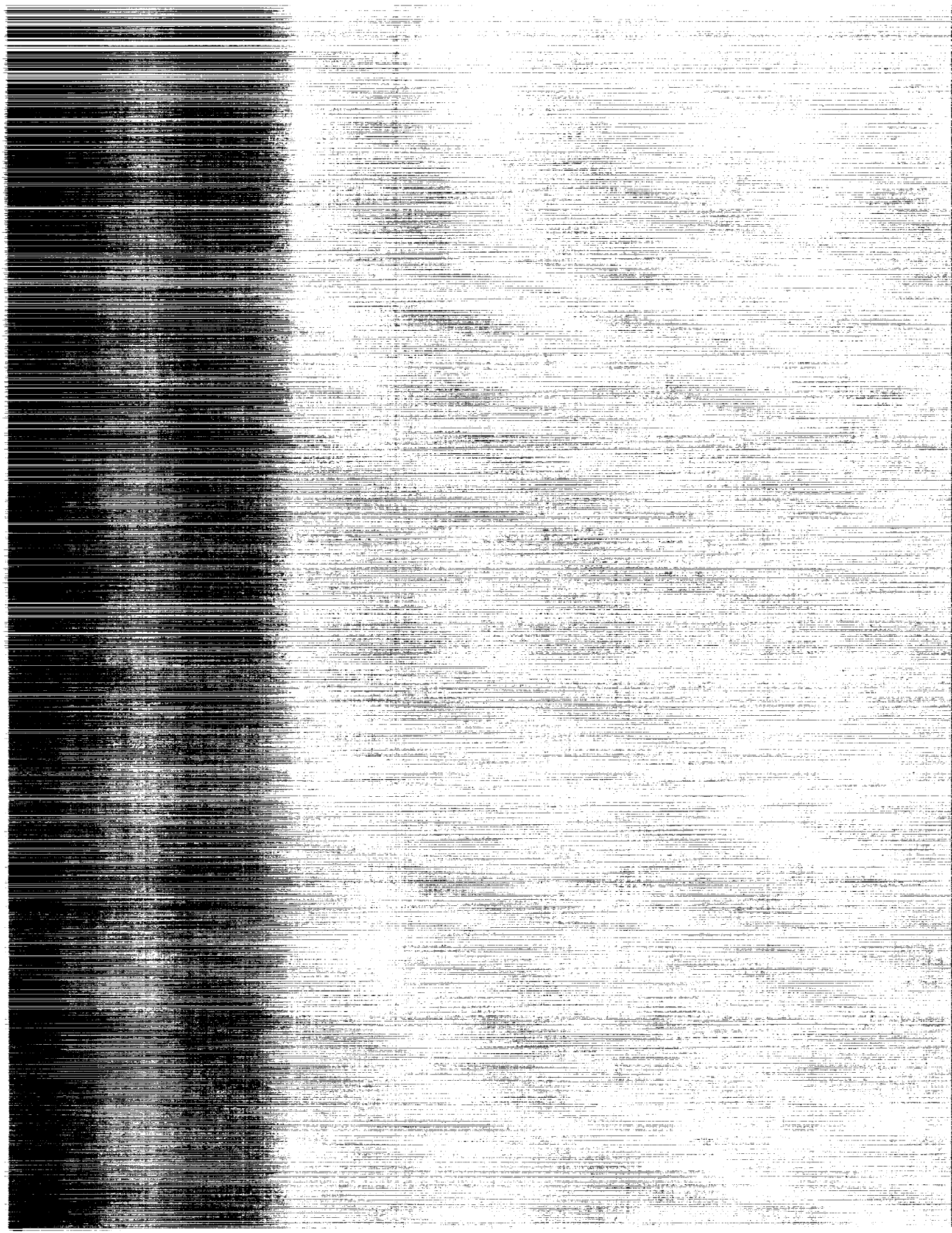
[10] R. Selby, A. Porter, D. Schmidt, and J. Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *Proc. 13th International Conference on Software Engineering*, pages 288–298, Austin, TX, May 1991.

[11] J. Tian, A. Porter, and M. V. Zelkowitz. An improved classification tree analysis of high cost modules based upon an axiomatic definition of complexity. In *Proc. 3rd International Symp. on Software Reliability Engineering*, Research Triangle Park, NC, October 1992.

[12] J. D. Valett. Automated support for experience-based software management. In *Proceedings of the Second Irvine Software Symposium (ISS '92)*, Irvine, CA, March 1992.

[13] A. von Mayrhauser. *Software Engineering: Methods and Management.* Academic Press, Inc., San Diego, CA, 1990.

SECTION 3 - SOFTWARE MEASUREMENT



SECTION 3—SOFTWARE MEASUREMENT

SMT

The technical paper included in this section was originally prepared as indicated below.

- "Measuring and Assessing Maintainability at the End of High Level Design," L. C. Briand, S. Morasca, and V. R. Basili, *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

Measuring and Assessing Maintainability at the End of High Level Design

Lionel C. Briand, Sandro Morasca, Victor R. Basili
Computer Science Department and Institute for Advanced Computer Studies
University of Maryland, College Park, MD, 20742

Abstract

Software architecture appears to be one of the main factors affecting software maintainability. Therefore, in order to be able to predict and assess maintainability early in the development process we need to be able to measure the high-level design characteristics that affect the change process. To this end, we propose a measurement approach, which is based on precise assumptions derived from the change process, which is based on Object-Oriented Design principles and is partially language independent. We define metrics for cohesion, coupling, and visibility in order to capture the difficulty of isolating, understanding, designing and validating changes.

1 Introduction

It has been shown that system architecture has an heavy impact on maintainability [R90, S90]. Numerous studies have attempted to capture the high-level design characteristics affecting the ease of maintenance of a software system [HK84, R87, S90]. Research in the field of design metrics [G86, SB91, Z91, AE92] has often been conducted according to a strategy intended to produce generic metrics assumed to be applicable in a variety of contexts and to many problem domains. However, such an approach has forced researchers to work without a clear framework and a well-defined goal. This frequently led to some degree of fuzziness in the metric definitions, properties, and underlying concepts, making the use of the metric difficult, the interpretation hazardous, and the results of the various validation studies somewhat contradictory [IS88, K88]. Some attempts were made to constrain the context of application to a particular programming language in order to come up with precisely and unambiguously defined metrics[AE92]. In other cases, the application domain of those metrics was restricted, e.g., error-prone subprograms [SB91], maintainability [R87]. In all cases (with the exception of [AE92], where these issues were partially addressed), no precise link was made between the studied process (e.g., change process) and the metrics, no clear and precisely defined assumptions were made about the process itself, and metrics were not defined by taking into account the specific issue to be addressed (e.g., maintainability).

We intentionally place ourselves in a well-defined framework (Ada [DoD83] and OOD[BO87]) and intend to focus exclusively on the change process during acceptance testing and maintenance, i.e., the change process performed by personnel who did not develop the software. Thereby, we propose more precise and effective high-level design metrics based on well-defined and verifiable assumptions which are closely related to the specific change process model instantiated at the NASA Goddard Space Flight Center. Thus, the applicability of those metrics is precisely defined, their validation easier, and their predictive ability more accurate. However, we also attempt to separate Ada specific concepts from language independent concepts in order to identify the part of the approach that is reusable for other programming languages.

Our goals can be expressed by using Basili's G/Q/M template [BR88]:

> Analyze the *high-level design of a software system* for the purpose of *prediction* with respect to *change difficulty* from the point of view of the *testers and maintainers.*
>
> Analyze the *high-level design of a software system* for the purpose of *evaluation* with respect to *change difficulty* from the point of view of the *designers.*

From a modeling perspective, our long-term goal is to be able to build models that predict change difficulty for the maintenance process, which will provide an early evaluation of maintainability, thus allowing better architectural/design decisions.

This paper first provides in Section 2 basic background information on the change process model and general definitions about the system constructs and the high-level design products. Section 3 presents the underlying concepts leading to the definitions of two basic metrics on top of which we define metrics for capturing module cohesion (Section 4), module coupling (Section 5), and coupling-based visibility control (Section

This work was supported in part by NASA grant NSG-5123, UMIACS, and NSF grant 01-5-24845

6). Finally, Section 7 summarizes the paper and presents the future directions of our research.

2 Background and Definitions

We first present the change process as perceived in our maintenance environment. Thus, we will be able to identify the various aspects of change difficulty (the *quality perspective* [BR88] of the goals of Section 1) and link our assumptions to this process so as to give a firm ground to our metrics. Then, we provide definitions for high-level design of a software system (the object of study of the goals of Section 1) and its basic constructs.

2.1 Change Process Model

In our environment of study (NASA Software Engineering Laboratory at the Goddard Space Flight Center), we view software maintenance as being composed of four primary phases, each encapsulating activities that may be performed concurrently, as shown in Figure 1.
There is a key milestone in the change process which is the decision of whether the change is going to be implemented or not. This is done based on a cost-benefit analysis after phase P1. The information necessary to this analysis is gathered during P1 and used for predicting the difficulty of designing, implementing and testing the change [BB92]. This information will encompass a description of the change itself and of the part of the system where the change is performed.

2.2 Object of Study

In the literature, there are two commonly accepted definitions of modules. The first one sees a module as a subprogram, and has been used in most of the design measurement publications [M77, CY79, HK84, R87, S90]. We choose the second category, which takes an object-oriented perspective, where a module is seen as a collection of routines, data and type definitions, i.e., a provider of computational services [BO87, G92].

Definition 1: Module.
A module is either a (possibly generic) subprogram, a (possibly generic) package, or a task. As such, a module comprises a specification and possibly a body.

Remark: Ada units vs. modules.
Compilation units are used in Ada for determining the compilation order and strategy. Instead, modules are defined here as Ada program units. We use the term module because it is a language independent concept. There are two kinds of Ada compilation units [DoD 83]: library units and secondary units. A library unit is either a package specification, a subprogram specification, or a whole subprogram which does not have a parent unit. Therefore, a library unit can be a module specification or a whole module. An Ada secondary unit is a unit with a parent unit and can only be a module body.

Definition 2: Data declaration.
A data declaration is either a type or an object (e.g., a constant, a variable, a formal parameter of a (possibly generic) subprogram or an entry, a generic formal object).

Definition 3: High-level Design product
The high-level design product is a collection of module specifications, either representing library units or belonging to secondary units, related by "uses" or "is a component of" [G92] relationships.

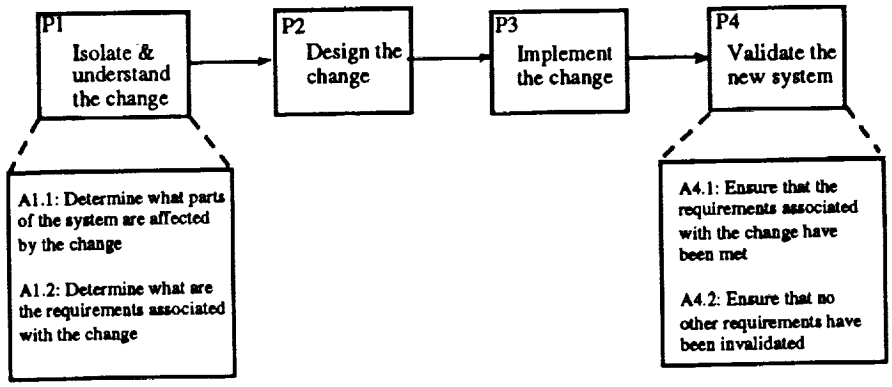


Figure 1. Change process

Since the remaining contents of units (e.g. local variables, algorithms) still remains to be determined in later design stages, our high-level design metrics will be mostly based on the information contained in module specifications. However, additional information to what is visible in the specifications may be available at the end of high-level design. For instance, given the specification of a module *m*, the designers have at least a rough idea of which objects declared in *m*'s and other modules' specification will be manipulated by a subprogram in *m*'s specification. It will be left to the person responsible for the metric program to decide whether or not it is worth collecting this kind of information, thus making the designer describe which global objects will be accessed by which subprograms or entries. For example, formatted comments might be a convenient way of conveying this information through module specifications and therefore of automating the collection of this type of information.

3 Interactions

We are looking for a primitive measure that links the change difficulty to the system design.
We therefore focus on the relationships that propagate side effects from data declarations to data declarations or subprograms when a change is performed. Those relationships will be called interactions and will be used to define metrics capturing cohesion and coupling within and between modules, respectively. Interactions linking subprograms to subprograms or data declarations will generally not be considered because they are encapsulated in module bodies and are therefore not detectable in our framework. However, these interactions are likely to be valuable although they will be rarely provided by the designer at the end of high-level design. For the sake of simplicity, we will not address this issue in the remainder of this paper. In all cases, these interactions will appear useful when looking at low-level design.

Definition 4: Data declaration-Data declaration (DD) Interaction.
A data declaration *A* DD-interacts with another data declaration *B* if a change in *A*'s declaration or use may cause the need for a change in *B*'s declaration or use .

The DD-interaction relationship is transitive. If *A* DD-interacts with *B*, and *B* DD-interacts with *C*, then a change in *A* may cause a change in *C*, i.e., *A* DD-interacts with *C*.

Data declarations can DD-interact with each other regardless of their location in the designed system. Therefore, the DD-interaction relationship can link data declarations belonging to the same module or to different modules.
By *DD-interactions(Dec_set1, Dec_set2)*, we will denote the number of DD-interactions from the set of data declarations *Dec_set1* to the set of data declarations *Dec_set2*.
At the end of high-level design, we may not have sufficient knowledge to understand with certainty whether there will be an interaction between two data declarations in the final software system, because we are not aware of all the DD-interactions present in the modules' bodies. On the basis of the information available from module specifications and their "uses" and "is a component of" relationships [G92], and from additional information provided by the designer, we can identify (1) the specification data declaration pairs that *are known* to DD-interact with each other, and (2) the specification data declaration pairs which *may* DD-interact with each other. We will say that there is an *actual DD-interaction* between data declaration pairs satisfying (1), and a *potential DD-interaction* between data declaration pairs satisfying (2). The latter kind of DD-interactions is only detectable by examining both specifications and bodies. Therefore, the set of actual DD-interactions is a subset of the set of potential DD-interactions.
The DD-interaction relationships can be defined in terms of the basic relationships between data declarations allowed by the language, which represent direct (i.e., not obtained by virtue of the transitivity of interaction relationships) DD-interactions. In Ada, data declaration *A* directly DD-interacts with data declaration *B* if *A* is used in *B*'s declaration or in a statement where *B* is assigned a value. As a consequence, as bodies are not available at high-level design time, we will only consider either the interactions detectable from the specifications or known by the designer.
DD-interactions provide a means to represent the relationships between individual data declarations. Yet, since procedures are not data declarations, DD-interactions *per se* are not able to capture the relationships between individual data declarations and subprograms, which are useful to understand whether data declarations and subprograms are related to each other and therefore should be encapsulated into the same module (see Section 4 on module cohesion).

Definition 5: Data declaration-Subprogram (DS) Interaction.
A data declaration DS-interacts with a subprogram if it DD-interacts with at least one of its data declarations.

Whenever a data declaration DD-interacts with *at least one* of the data declarations contained in a subprogram specification, the DS-interaction relationship between the data declaration and the subprogram can be detected by examining the high-level design. For instance, from the code fragment in Figure 2, it is apparent that both type *T1* and object *OBJECT11* DS-interact with procedure *P11*, since they both DD-interact with parameter *PAR11*, one of procedure *P11*'s specification data declarations.

```
package Pk1 is
  ...
  type T1 is ...;
  OBJECT11, OBJECT12: T1;
  procedure P11(PAR11: in T1:=OBJECT11);
  ...
  package Pk2 is
        ...
        OBJECT13: T1;
        type T2 is array (1..100) of T1;
        OBJECT21: T2;
        procedure P21(PAR21: in out T2);
        ...
  end Pk2;

  task Tk is
        entry E1(PAR12: in out T1);
        entry E2(PAR22: in out T2);
  end Tk;
  ...
  OBJECT22: Pk2.T2;
  ...
end Pk1;
```

Figure 2. Program fragment

On the other hand, there may be DS-interactions that are not detectable only on the basis of the Ada code representing the high-level design, since they are due to DS-interactions occuring in subprogram bodies. For instance, from the code fragment above, we cannot tell whether *OBJECT12* DS-interacts (as a global variable) with procedure *P11*. The designers may very likely be able to supply this additional piece of information. More specifically, the designers can answer in three different ways:

(1) *OBJECT12* DS-interact with *P11*
(2) *OBJECT12* does not DS-interact with *P11*
(3) the information they have is not sufficient

It is worth saying that answers of kind (2) provide valuable, though negative, information on the DS-interaction present in a system.

Remark:
Definition 5 states that DS-interaction is a relationship between data declarations and a subprogram, which is a specific kind of module. Since we are interested in the interactions between data declarations and algorithms, we did not provide a more comprehensive definition also accounting for the relationships between a data declaration and a package or a task, which are the other possible kinds of module. As a matter of fact,

- packages are a means for grouping/encapsulating data declarations and subprograms (and possibly tasks and other packages). Therefore, we will not examine the relationships between a data declaration and a package as a whole.

- tasks are defined in terms of their entries, i.e., they can be seen as a collection of entries, which we will see as a particular kind of subprograms. Therefore, we will not examine the relationships between a data declaration and a task as a whole.

For graphical convenience, both sets of interaction relationships will be represented by directed graphs, the *DD-interaction graph*, and the *DS-interaction graph*, respectively. In both graphs (see Figures 3 and 4, which respectively represent DD- and DS-interaction graphs for the code fragment of Figure 2), data declarations are represented by rounded nodes, subprograms by thick lined boxes, and packages and tasks by thin lined boxes. Solid arcs represent interactions that can be known by either inspecting the high-level design or collecting information from the designers, dashed arcs represent those interactions that are not detectable from the high-level design and that will not occur in the body, according to the designers' opinion. (For simplicity's sake, in Figure 3 we only represent *direct* DD-interactions.) For instance, the existence of an DD-interaction between object *OBJECT12* and *PAR11* and the lack of interaction between *OBJECT13* and *PAR21* have been signaled by the designer. Since this information may improve significantly the accuracy of the count of DS-interactions and is in many cases known by the designers, we strongly recommend that the reader pay attention to this issue.

Our approach to design measurement and evaluation will be based on the above definitions and will be guided by the general principle that system architecture should have low average module coupling and high average cohesion. This is assumed to improve the capability of a system to be decomposed in highly independent and easy to understand pieces. Cohesion captures the extent to which the data declarations and subprograms that interact are grouped within the same modules, whereas coupling captures their dispersion by

looking at module dependencies and exports. These issues are addressed in the next sections.

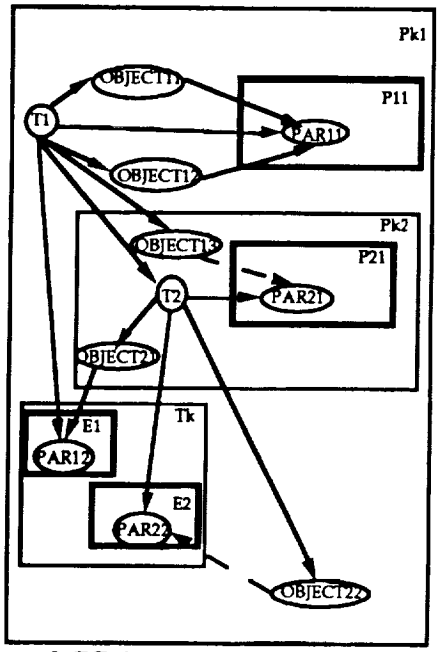


Figure 3. DD-interaction graph for the program fragment in Figure 2

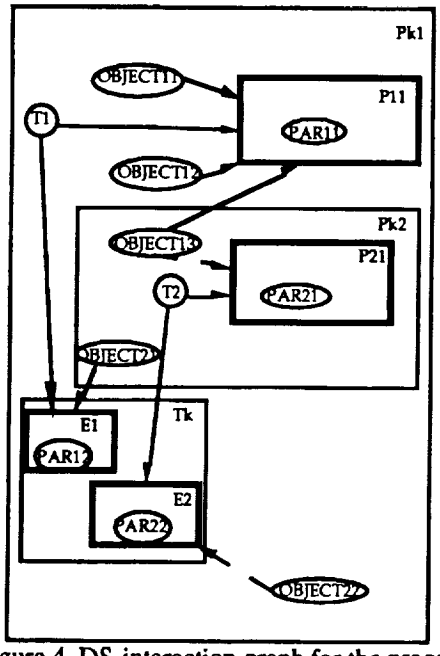


Figure 4. DS-interaction graph for the program fragment in Figure 2

4 Module Cohesion

It is generally acknowledged that a high degree of cohesion is a desirable property of a module. Here, after a general definition for cohesion, we provide assumptions to restrict it to our specific viewpoint—change. This allows the definition of change-oriented cohesion metrics which are also based on our OOD definition of module.

4.1 Definitions

Definition 6: Cohesion (CH)
Cohesion is the extent to which a module only contains data declarations and subprograms which are conceptually related to each other.

Assumption A-CH:
From our "change process" viewpoint, a high degree of cohesion is desirable because information relevant to a particular change within a module should not be scattered among irrelevant information. Data declarations and subprograms which are not related to each other should be encapsulated to the extent possible into different modules. We believe that this issue is especially important for activity *A1.2* (see Figure 1) where the change requirements have to be understood.

4.2 Cohesive Interactions

Since we place ourselves at the end of high-level design and we want to look at the set of services provided by a module, we are interested in evaluating how tight are the relationships between the data declarations declared within a module specification, and between the data declarations and the subprograms declared there. We will capture this by means of cohesive interactions.

Definition 6: Cohesive Interaction.
The set of cohesive interactions in a module is the union of the sets of DS-interactions and DD-interactions, with the exception of those DD-interactions between a data declaration and a subprogram formal parameter.

We do not consider the DD-interactions linking a data declaration to a subprogram parameter as relevant to cohesion, since they are already accounted for by DS-interactions and we are interested in evaluating the degree of cohesion between data declarations (data), and procedures (algorithms) seen as a whole.

Remark.
It is worth reminding the reader that those relationships that cannot be detected by inspecting the specifications, i.e., global variables interacting with subprogram bodies, can actually be quite relevant to cohesion evaluation, because they often represent the connections between an object and

the subprograms that access it; such connections are the relationships that make an abstract object cohesive.

4.3 Cohesion Metrics

Based upon the above definition of cohesive interactions, we define a cohesion metric that satisfies the following two properties.

Property 1: Normalization.
Given a module m, the metric $cohesion(m)$ belongs to the interval [0,1].

Normalization allows meaningful comparisons between the cohesions of different modules, since they all belong to the same interval.

Property 2: Monotonicity.
Let m_1 be a module and CI_1 its set of cohesive interactions. If m_2 is a modified version of m_1 with one more cohesive interaction so that CI_2 includes CI_1, then $cohesion(m_2) \geq cohesion(m_1)$.

Since there is uncertainty on the DD- and DS-interactions present in a module, due to the incompleteness of the information that can be collected from the specifications and the designers, we define not only a metric but the boundaries of an uncertainty interval.

Definition 7: Ratios of Cohesive Interactions.

Neutral Ratio of Cohesive Interactions (NRCI):
All unknown CIs are not taken into account

NRCI=#knownCIs/(#potentialCIs-#unknownCIs)

Pessimistic Ratio of Cohesive Interactions (PRCI):
All unknown CIs are considered as if they where known not to be actual interactions.

PRCI = #knownCIs/#potentialCIs

Optimistic Ratio of Cohesive Interactions (ORCI):
All unknown CIs are considered as if they where known to be actual interactions

ORCI=(#knownCIs #unknownCIs)/#potentialCIs

If PRCI, NRCI, and ORCI are all not undefined, it can be shown that

PRCI ≤ NRCI ≤ ORCI

Figure 5 shows representative and interesting examples of module cohesion computation. Each thin lined box represents a module specification. T's, O's, and SP's will characterize types, objects and subprograms, respectively. We did not represent procedure parameters, since they do not belong to any cohesive interaction, nor packages nor tasks, since they are inessential to our discussion. However, we represented all direct and transitive interactions.

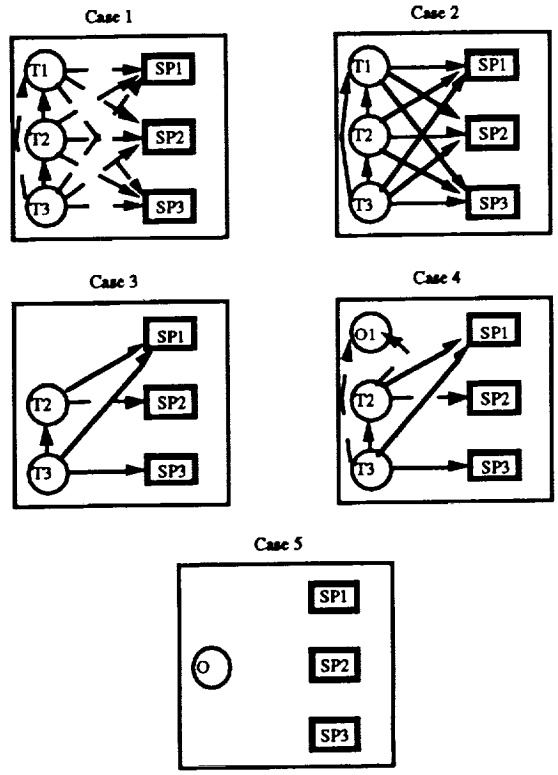


Figure 5: Cohesion examples

Case 1: No cohesive interaction is present

PRCI = 0/12 = 0
NRCI = 0/12 = 0
ORCI = 0/12 = 0

Case 2: All possible cohesive interactions are present

PRCI = 12/12 = 1
NRCI = 12/12 = 1
ORCI = 12/12 = 1

Case 3: Incomplete interaction graph

PRCI = 4/7 = .571
NRCI = 4/5 = .8
ORCI = 6/7 = .857

Case 4: Isolated object
O1 has been added to Case 3. This decreases cohesion because O1 has no known interactions with the rest of the module data declarations and subprograms.

PRCI = 4/12 = .333
NRCI = 4/7 = .571
ORCI = 9/12 = .75

Case 5: Single object

PRCI = 0/3 = 0
NRCI (= 0/0) = undefined
ORCI = 3/3 = 1

No information is available on the interactions between object O and the three subprograms. Therefore, ORCI and PRCI provide the bounds of the admissible range for cohesion, and NRCI is undefined, i.e., it could take any value in between. The more incomplete the information, the wider the uncertainty interval.

5 Module Coupling

According to commonly accepted design principles, design must show low coupling between modules. In this section, we first give general definitions and assumptions on coupling (Section 5.1). Then, we present a set of metrics (Section 5.2), and discuss the issue of genericity (Section 5.3) in the context of coupling.

5.1 Definitions

Definition 8: Import Coupling of a module (IC):
Import Coupling is the extent to which a module depends on imported external data declarations.

Assumption A-IC:
The more dependent a module on external data declarations, the more difficult it is to understand in isolation. In other words, the larger the amount of external data declarations, the more incomplete the local description of the module specification, the more spread the information necessary to isolate and understand a change. Thus, if there is a high average coupling within a set of modules, both activities *A1.1* and *A1.2* in Figure 1 are affected. The design of the change (phase *P2* in Figure 1) is also more complex.

Definition 9: Export Coupling of a module (EC).
Export coupling is the extent to which a module's internal data declarations affect the data declarations of the other modules in the system.

Assumption A-EC:
Export coupling is related to how a particular module is used in the system. As such, EC should have a direct impact on understanding the effect of a change on the rest of the system, and on validating the system after the change.
The larger the number of DD-interactions with external data declarations, the larger the likelihood of ripple effects when a change is implemented (activity *A4.2* in Figure 1). Also, the larger the number of potential DD-interactions, the more complex testing and verification become, since potential side effects have to be identified and addressed based on actual DD-interactions (activities *A4.1* and *A4.2* in Figure 1).

The import coupling of a module will be expressed in terms of the actual DD-interactions between imported/visible external data declarations (i.e. global) and the internal data declarations of the module. Export coupling will be based on both the actual and potential DD-interactions between locally defined data declarations and the other data declarations within the scope of the module. Actual DD-interactions are important because they capture the actual dependencies between a module and its context of declaration and therefore should be closely related to the likelihood of ripple effects. According to the defined assumption, the number of potential DD-interactions of a module with its context of declaration should be related to the ease of verifying and testing the side effects of the implemented change. These potential DD-interactions will simply be determined by the programming language visibility rules.

5.2 Metrics Based on Coupling

The issue will be first addressed by ignoring generic modules for the sake of simplification. Generic modules and their impact on the defined metrics will be treated in Section 5.3.

Definition 10: Global versus Locally defined data declarations
We will denote by *Global(m)* the set of all the external data declarations imported by a module *m*, and by *Local(m)* the set of all the locally defined data declarations in module *m*.

Definition 11: Scope of a module
Scope(m) is the set of all data declarations declared outside the module for which the internal data declarations of module *m* are visible.

Definition 12: Import Coupling
We will use the following metric to capture Import Coupling

IC(m) = DD-interactions(Global(m), Local(m))

In the above definition, we have considered all sort of imports equally. However, in terms of impact on the change difficulty in a particular module, imports from the same hierarchy or the same subsystem do not equate with imports from outside the module's hierarchy or subsystem. There are several reasons for this, e.g., people may have a better familiarity with the subsystem they are in charge of maintaining, understanding a module in another hierarchy increases the load of information to be known for understanding the change. Although we do not fully investigate this complex issue here, a simple solution to refine *IC* could be to define all the metrics presented below separately for several categories of coupling, e.g., coupling with modules outside the subsystem.
Each box in Figure 6 represents a module specification. Submodule specifications *C2* and *C3* are located in their parent's body *C1*. *C2* is assumed to be declared before *C3* and therefore visible to *C3*. The *Inst()*, *Sub()*, *Derived()*, *ValDep()* and *Const()* functions specify if one data declaration is respectively the object instantiation of a type, a subtype of a type, the derived type of another type, an object dependent on the value of another object (e.g. initialization), an object used to constrain a type or another object definition. Note that the same data declaration may interact with several data declarations, e.g., *T21* in Figure 6. *Tij* and *Oij* data declarations represent respectively types and objects in module *Ci*. *FPij* represents subprogram formal parameters. Even though they are objects, we identified them by a different symbol in order to improve the figure readability. The *IC* values for the modules in Figure 6 are computed as follows

IC(m) = direct DD-interactions + transitive DD-interactions

IC(C1) = 0 + 0 = 0
IC(C2) = 3 + 1 = 4
-- from C1 (direct: O11 twice,T12; transitive: T21)
IC(C3) = 2 + 2 = 4
-- from C1 (direct: T12; transitive: T12 twice) and C2 (direct: T21)
IC(C4) = 1 + 0 = 1 -- from C1 (direct: T11)

Definition 13: Potential and Actual Export Coupling
As presented in the assumption *A-EC*, both actual and potential coupling need to be measured.

EC-Actual(m) = DD-interactions(Local(m), Scope(m))

EC-Potential(m) = |Local(m)| · |Scope(m)|

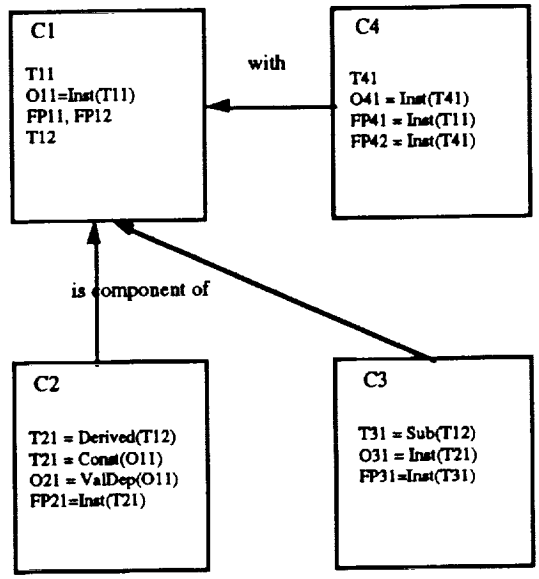


Figure 6: Calculation of IC and EC with non-generic components only

In the example of Figure 6, illustrated by the results presented below, we see that *C1* expectedly shows the largest actual and potential export coupling.

EC-Actual(m) = direct DD-interactions + transitive DD-interactions

EC-Actual(C1) = 5 + 3 = 8
-- to C2 (direct: T12, O11 twice; transitive: T12),
-- to C3 (direct: T12; transitive: T12 twice),
-- to C4 (direct: T11)
EC-Actual(C2) = 1 + 1 = 2
-- to C3 (direct: T21; transitive: T21)
EC-Actual(C3) = 0 + 0 = 0
EC-Actual(C4) = 0 + 0 = 0

EC-Potential(C1) = 5 · 10 = 50
EC-Potential(C2) = 3 · 3 = 9
EC-Potential(C3) = 3 · 0 = 0
EC-Potential(C4) = 3 · 0 = 0

We now introduce a normalized measure, Relative Dependency, to capture how dependent a module is on external data declarations with respect to the whole set of data declarations it can access, i.e., the external data declarations and its own data declarations. This normalized measure may contribute to capture the difficulty of the

understanding process described in assumption *A-IC*, along with the absolute *IC* measure.

Definition 14: Relative Dependency (RD)
The relative dependency of a module *m* is the ratio of Import Coupling normalized by the total number of DD-interactions, i.e., within *m* itself and between the data declarations external to m and m.

$$RD(m) = IC(m)/(DD\text{-}interactions(Local(m), Local(m)) + IC(m))$$

RD(m) is therefore a unitless measure of import coupling of the module with the rest of the system which is relative to the total number of DD-interactions. Thus, a large module with a large import coupling might show a somewhat low relative dependency.

For Figure 6, we obtain the following results:

DD-interactions(C1, C1) = 1
DD-interactions(C2, C2) = 1
DD-interactions(C3, C3) = 1
DD-interactions(C4, C4) = 2

RD(C1) = 0/(1+0) = 0
RD(C2) = 4/(1+4) = 0.8
RD(C3) = 4/(1+4) = 0.8
RD(C4) = 1/(2+1) = 0.33

We can differentiate two main families of modules, based on IC and EC: "servers", i.e., provider of services, and "clients", i.e., users of services.

Definition 15: Coupling type (CT)
The coupling type of a module *m* is the ratio of Import Coupling normalized by the total Export and Import Coupling of module *m*.

$$CT(m) = IC(m)/(EC\text{-}Actual(m) + IC(m))$$

When *CT < 0.5*, then the module is more of the type "server"; otherwise, it can be classified as a "client". The first type of modules is expected to be more often at the top of the system hierarchies while the second type should be more common at the bottom of those hierarchies. This is what happens in the example in Figure 6, as the results presented below show.

CT(C1) = 0/(8+0) = 0 -- server
CT(C2) = 4/(2+4) = 0.66 -- more of the client type
CT(C3) = 4/(0+4) = 1 -- client
CT(C4) = 1/(0+1) = 1 -- client

Exceptions to this pattern may be the symptom of anomalies in system design.

5.3 The Treatment of Generic Modules

There are two possible ways of taking into account generics when calculating coupling. Either each instance may be seen as a different module or a generic may be seen as any other module whose scope/global data declarations is/are the union of the scope/global data declarations of its instances. The second solution does not consider instances as independent modules and appears to be more suitable to our specific perspective (i.e., the change process) since instances cannot be modified directly and only one module is to be maintained: the generic module. In other words, if *N* instances are generated, we will not count coupling as if *N* modules were actually developed since those instances may only undertake change through their corresponding generic module. Generic formal parameters allow for the substitution of objects, types and subprograms. This substitution does not have any impact on the number and the kind of exported data declarations (i.e. same number of type, object declarations respectively imported and exported).
When calculating import coupling, we will count the DD-interactions of the generic modules with the union of the global data declarations specific to their instances. When calculating export coupling, we will count the DD-interactions of the generic modules within the union of the scope of their respective instances. Consistent with the definition of DD-interaction, generic formal parameters DD-interact with their particular generic actual parameters (i.e. type, object) when the generic module is instantiated since a change in the former may imply a change in the latter.
This is what the following example illustrates. The graphical formalism is identical to the one used in Figure 6 and function *New(G, P)* represents a new instantiation of a generic package or subprogram *G* with a generic formal parameter *GFP1* and its generic actual parameter set *{P1, P2}*.
C2 and *C3* only import data declarations from *G* (**with** *TG1*). *C1* imports from *G* (*P1*, *P2* DD-interact with *FGP1*).

IC(m) = direct DD-interactions + transitive DD-interactions

IC(C1) = 2 + 0 = 2 -- from G
IC(C2) = 2 + 1 = 3 -- from G and C1
IC(C3) = 3 + 1 = 4 -- from G and C1
IC(G) = 0 + 0 = 0

EC(m) = direct DD-interactions + transitive DD-interactions

EC-Actual(C1) = 2 + 2 = 4 -- to C2, C3
EC-Actual(C2) = 0 + 0 = 0
EC-Actual(C3) = 0 + 0 = 0
EC-Actual(G) = 5 + 0 = 5 -- to C1, C2, C3

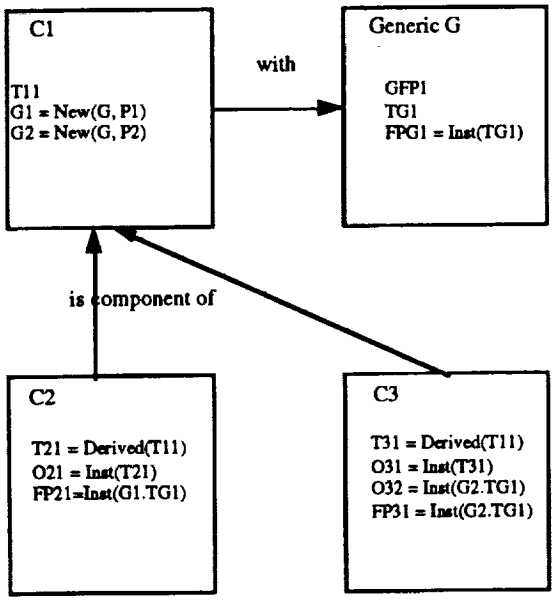


Figure 7: Generics when calculating coupling

The RD metric shows that G is the only fully independent module. The others strongly depend on external data declarations:

DD-interactions(C1) = 0
DD-interactions(C2) = 1
DD-interactions(C3) = 1
DD-interactions(G) = 1

RD(C1) = 2 / (0 + 2) = 1
RD(C2) = 3 / (1 + 3) = 0.75
RD(C3) = 4 / (1 + 4) = 0.8
RD(G) = 0 / (1 + 0) = 0

6 A Visibility Control Metric Based on Coupling

As opposed to the metrics presented in previous sections, this metric does not characterize modules but sets of modules. Here, we want to assess to which extent visibility is controlled in the design of a system, subsystems or any system part [G86, AE92]. Thus, we want to identify design flaws related to visibility.

Assumption A-VC:

If the system, the subsystem or the hierarchy has been designed by following minimal visibility rules, modules with larger potential export coupling should also have larger actual export coupling. This is the case in the above example where the ranking according to *EC-Potential* is identical to the ranking by *EC-Actual*. Therefore, we want to measure the correlation between *EC-Actual* and *EC-Potential* in order to determine whether or not highly visible modules are also highly used modules. In other words, this can be interpreted as how well visibility is controlled within the system or a part thereof.

Remark.
We do not intend to judge the designer work through this process, since other constraints may bias the design towards a non optimal visibility control. We look at it from the narrow perspective of the change process, leaving to the designer the decision of possible tradeoffs between maintainability and other criteria, e.g., performance.

We do not want the measure of correlation to be based on parametric assumptions since we do not know what kind of relationship to expect between actual and potential export coupling[CAP88]. One way of doing it is to use a non-parametric statistic which takes into account the rank of each module with respect to both *EC-potential* and *EC-actual*. This type of statistic does not require any functional assumption and is moreover robust to outliers. Thus, we will be protected against illusory strong correlations due to outliers and falsely weak correlations due to wrong functional assumptions. If visibility is close to minimal, we assume the ranks of the modules to be similar with respect to those two metrics.

Definition 16: Visibility control (VC)
The visibility control of a set of modules *SM* (*VC(SM)*) is measured by means of the Spearman's rank correlation coefficient [CAP88] between the actual Export Coupling and the potential Export Coupling

$$VC(SM)=1-[\Sigma_{m\in SM}(D^2(m))/(|SM|(|SM|^2-1)/6)]$$

where D(m) = Rank(EC - Actual(m)) - Rank(EC-Potential(m))

The larger *VC(SM)*, the closer to minimal the visibility. When there is no association, it can be

shown that $\Sigma_{m \in SM}(Distance^2(m)) = (|SM|(|SM|^2 - 1)/6)$, so $VC(SM) = 0$.

7 Conclusions

In this paper, we have presented a comprehensive approach for evaluating the high-level design of software systems which is summarized by the following characteristics:

- early available metrics based on precisely defined assumptions and related without ambiguity to the defined change process model
- definitions of module cohesion, module coupling and visibility control consistently based on the notion of interaction, which is closely related to the phenomenon of change side effects
- an OOD [BO87] view of a software module as opposed to the usual subroutine perspective [M77, YC79] of coupling and cohesion evaluation
- a clear separation between Ada-specific and language-independent concepts.

Our future research will encompass:

- the definition and refinement of other higher-level metrics based on module coupling and cohesion that will characterize higher-level constructs, e.g., module hierarchies, subsystems.
- the experimental validation of the proposed metrics with respect to change difficulty (i.e., man-hours) and size (i.e., number of modules changed, lines of code removed, changed, added).
- the development of high level metrics based on other software engineering principles, such as information hiding and reuse.

Acknowledgments

We thank Bill Thomas and Chris Hetmanski for their helpful comments on the earlier drafts of this paper.

References

[AE92] W. Agresti and W. Evanco, "Projecting Software Defects from Analyzing Ada Designs", *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[BB92] L. Briand, V. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process", Conference on Software Maintenance, 1992, Orlando, Florida.

[BO87] G. Booch, "Software Engineering with Ada", Benjamin/Cumming Publishing Company, Inc., Menlo Park, California, 1987.

[BR88] V. Basili and H. Rombach,"The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

[CAP88] J. Capon, "Elementary Statistics", "Statistics for the Social Sciences", Wadsworth Publishing Company, 1988.

[CY79] E. Yourdon, L. Constantine, "Structured Design", Prentice Hall, 1979

[DoD83] ANSI/MIL-STD-1815A-1983, Reference Manual of the Ada Programming Languages, U.S. Department of Defense, 1983

[G86] J. Gannon, E. Katz, V. Basili, "Metrics for Ada Packages: an Initial Study", Communications of the ACM, Vol. 29, N. 7, July 1986.

[G92] C. Ghezzi, M. Jazayeri, D. Mandrioli, "Fundamentals of Software Engineering", Prentice Hall, Englewood Cliffs, NJ, 1992

[HK84] S. Henry, D. Kafura, "The Evaluation of Systems' Structure Using Quantitative Metrics", Software Practice and Experience, 14 (6), June, 1984.

[IS88] D. Ince, M. Shepperd, "System Design Metrics: a Review and Perspective", Proc. Software Engineering 88, pages 23-27, 1988

[K88] B. Kitchenham, "An Evaluation of Software Structure Metrics", Proc. COMPSAC 88, 1988

[M77] J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, 12(10):61-64, 1977

[R87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure and Maintainability:", *IEEE Trans. Software Eng.*, 13 (5), May, 1987.

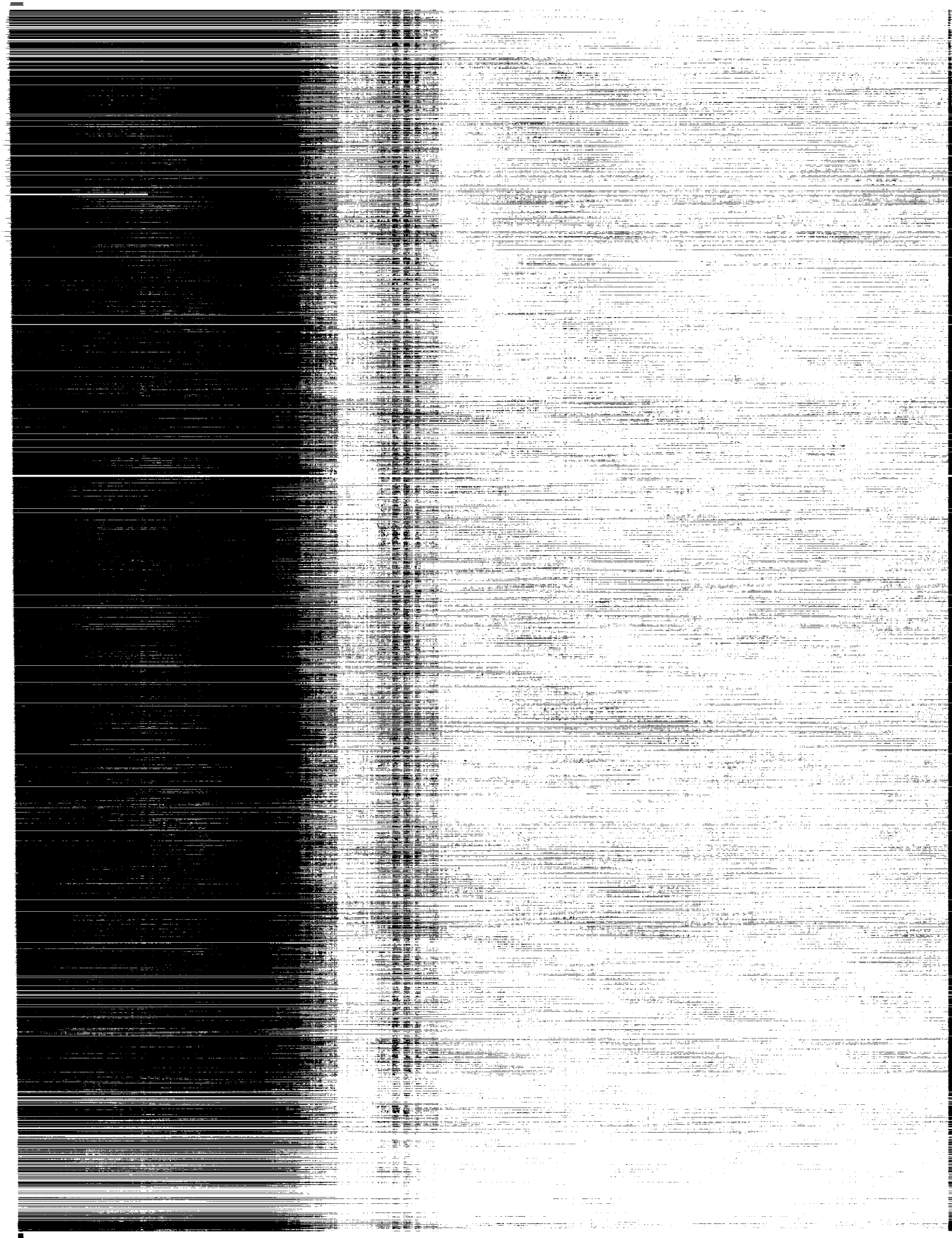
[R90] H. D. Rombach, "Design Measurement: Some Lessons Learned", *IEEE Software*, March 1990.

[S90] M. Shepperd, "Design Metrics: An Empirical Analysis", *Software Engineering Journal*, January 1990.

[SB91] R. Selby and V. Basili, "Analyzing Error-Prone System Structure", *IEEE Trans. Software Eng.*, 17 (2), February, 1991.

[Z91] W. Zage, D. Zage, P. McDaniel, I. Khan, "Evaluating Design Metrics on Large-Scale Software", SERC-TR-106-P, September 1991.

SECTION 4 - TECHNOLOGY EVALUATIONS



SECTION 4—TECHNOLOGY EVALUATIONS

omit

The technical paper included in this section was originally prepared as indicated below.

- "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," M. Stark, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

IMPACTS OF OBJECT-ORIENTED TECHNOLOGIES: SEVEN YEARS OF SEL STUDIES

Mike Stark

SOFTWARE ENGINEERING BRANCH
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771
(301) 286-5048

ABSTRACT

This paper examines the premise that object-oriented technology (OOT) is the most significant technology ever examined by the Software Engineering Laboratory. The evolution of the use of OOT in the Software Engineering Laboratory (SEL) "Experience Factory" is described in terms of the SEL's original expectations, focusing on how successive generations of projects have used OOT. General conclusions are drawn on how the usage of the technology has evolved in this environment.

INTRODUCTION

The Software Engineering Laboratory (SEL) sponsored by the National Aeronautics and Space Administration/ Goddard Space Flight Center (NASA/ GSFC), has three primary organizational members: the Software Engineering Branch of NASA/GSFC, the Department of Computer Science of the University of Maryland, and the Software Engineering Operation of Computer Sciences Corporation. It was created in 1976 to investigate the effectiveness of software engineering technologies applied to the development of applications software. As it seeks to understand the software development process in the GSFC environment, the SEL measures the effects of various methodologies, tools, and models against a baseline derived from current development practices.

In the SEL production environment, the language usage is approximately 70 percent FORTRAN, 15 percent Ada, and 15 percent C. This is in contrast to the almost 100-percent FORTRAN environment in 1985. Projects typically last between two and four years, and they range in size from 100,000 to 300,000 source lines of code (SLOC). A typical project consists of between 20 percent and 30 percent code reused from previous projects.

The SEL has examined many technologies, some of which have major effects on how software is developed in the SEL production environment, where ground-support software is produced for the Flight Dynamics Division (FDD) at Goddard Spaceflight Center (GSFC). One technology, Object-Oriented Technology (OOT), has attracted special notice in recent years, causing Frank McGarry, head of Goddard's Software Engineering

Branch, to remark a year ago that "Object-Oriented Technology may be the most influential method studied by the SEL to date" (Reference 1).

THE EXPECTATIONS AND REALITY OF OOT

The development of highly reusable software is one of the promises of OOT. The initial expectation for OOT was that this increased reuse would yield benefits in the cost and the reliability of software products. In addition, it was expected that OOT would be more intuitive than the structured development traditionally used in this environment, making the development process more efficient. Therefore, the SEL expected that, in addition to the reuse benefits, the cost of developing new code would also decrease.

The specific measures applied to assess the effect of OOT include cost in hours per thousand source lines of code (KSLOC), reliability by measuring errors per KSLOC, and the duration of the project in months. To date, OOT has been applied on eleven projects in the SEL. These projects can be grouped into three families of completed projects and an ongoing effort to develop generalized flight dynamics application software.

The completed projects (Figure 1) include three early Ada simulators built between 1985 and 1988, as well as three FORTRAN ground-support systems developed from the Multimission Three-Axis Attitude Support System (MTASS) and four telemetry simulators developed from multimission simulator code, all of which multimission applications were developed between 1988 and 1991.

During the seven years the SEL has been experimenting with OOT, developers have gained more understanding of which object-oriented concepts are most applicable in the FDD environment. The most important part of the evolution is the application of object-oriented concepts to a greater portion of the development life cycle over time. The knowledge gained during the development of these three families of systems is being applied in the development of generalized flight dynamics applications.

Despite its later appearance chronologically, the MTASS family of systems (Figure 2) should be

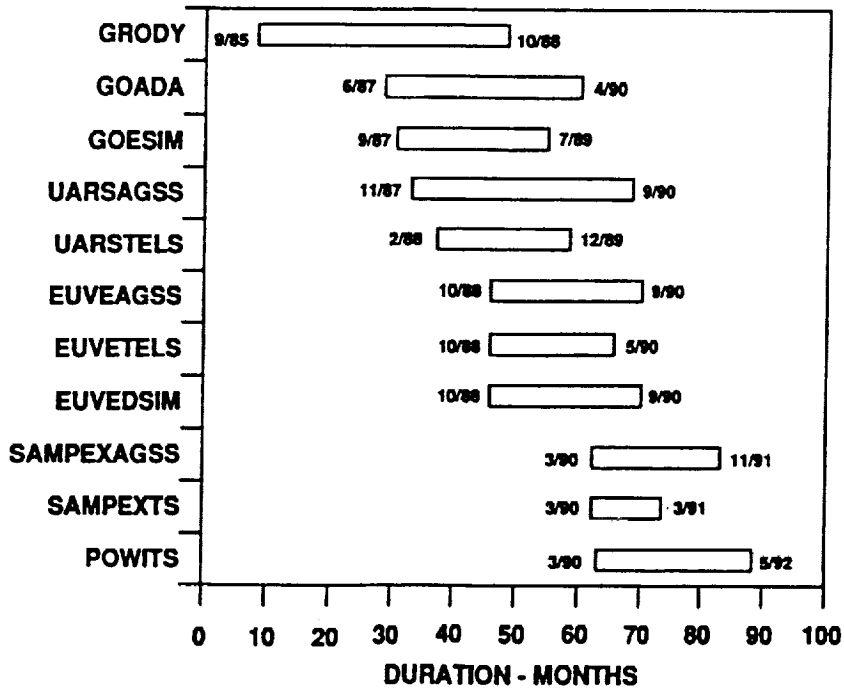


Figure 1. Projects Using Object-Oriented Technology

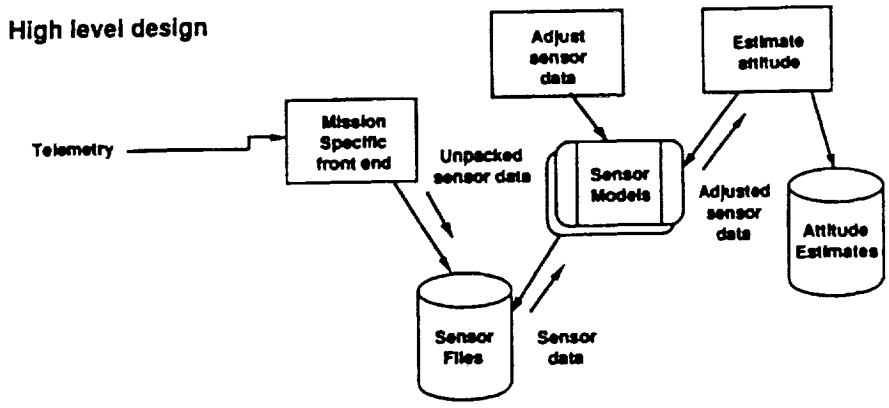


Figure 2. MTASS Design

examined first because it represents a modest infusion of OOT. MTASS started with a ground-support system that was developed as a common system for two different satellites, the Upper Atmosphere Research Satellite (UARS) and the Extreme UltraViolet Explorer (EUVE) satellite. It was then reused for the Solar, Anomalous, and Magnetosphere Particle Explorer (SAMPEX).

All ground-support systems read in telemetry and produce attitude (spacecraft orientation) estimates. The difference is that, where previous systems had stored all sensor data in one file specifically designed for the mission, MTASS developed separate interface routines and file formats for each kind of sensor. Only one mission-specific, front-end telemetry processor had to be developed for each new mission.

This basic grouping of data and of operations on the data is the most important object-oriented concept in the FDD environment. This change alone increased code reuse from the baseline 20 percent to 30 percent to around 75 percent or 80 percent.

It should be emphasized that the use of OOT on these projects was modest. The implementation language is FORTRAN, and the standard structured design notation was used to document the system. The object-orientation of the sensor model design was recognized during coding rather than consciously planned during design. Nonetheless, this one simple concept has had tremendous benefit in developing ground-support software faster and at a lower cost.

The earliest purposeful use of object-orientation in the SEL environment was associated with the introduction of Ada in 1985. The first Ada project, the Gamma Ray Observatory (GRO) Dynamics Simulator in Ada (GRODY), was developed as an experiment in parallel with an operational FORTRAN simulator. Previous Ada experiments (Reference 2) had produced designs and code that looked like Ada versions of FORTRAN systems. To avoid this, the GRODY team was trained in a variety of design methods, including Booch's Object-Oriented Design (OOD) method (Reference 3), stepwise refinement, and process abstraction. In addition, one of the team members had an academic background in OOD.

OOD emerged as a clear favorite, but in early 1985 Booch's method was not mature enough to support large production projects. Stark and Seidewitz developed the General Object-Oriented Design (GOOD) method during the GRODY project to meet these needs (Reference 4). Its first application was on the Geostationary Operational Environmental Satellite (GOES) Dynamics Simulator in Ada (GOADA), a project started in 1987. The GOES Telemetry Simulator (GOESIM) was also implemented in Ada. GOESIM was developed using structured design techniques, although GRODY packages designed with an object-oriented approach were reused on GOESIM.

The goal of the early Ada simulation projects was to learn the appropriate use of the Ada language, with a view towards increasing software reuse. Other goals were considered less important. The GRODY team, for example, was specifically instructed not

to worry about the real-time requirement being imposed on the FORTRAN simulator, and in fact GOADA was able to achieve higher than usual reuse from GRODY code. However, the lack of attention to performance led to systems with disappointing performance.

The SEL responded to this issue by studying the performance of the GOADA simulator in detail to determine if the performance problems were caused by the Ada language, the OOD concept, or by the GOADA design itself. The studies estimated the effect of various improvements on the execution speed of a simulation. These improvements included changes such as removing repeated inversion of the same matrix from an integrators derivative function or simplifying the internal data structure of an objects state. The inefficiencies were not caused by the use of object oriented technologies, and improving the performance with these corrections would not compromise the object-oriented design. Figure 3 shows that making all these changes to the full simulator would improve performance to the levels attained by similar FORTRAN simulators.

The next generation of projects is a multimission telemetry simulation architecture, built around Ada generic packages. Figure 4 shows how two sensor models use a generic sensor package for common functions such as writing reports and simulated data files. Here, each sensor has its own specific modeling procedure that is used to instantiate the generic. In addition, these model procedures are built around other generics that provide common functionality such as modeling sensor failures or digitizing simulated sensor data. The arrows indicate dependencies between software modules. For example, the Gyro object depends on procedure Gyro_Model to provide gyro specific functionality, and it instantiates the Generic Sensor package to provide more general sensor capabilities. One of the interesting consequences of the extensive use of generics is that the system size decreased; the previous generation of Ada telemetry simulator contained 92 KSLOC, but this multimission simulator contains only 69 KSLOC.

This architecture was the first simulator designed to facilitate reuse from mission to mission. Unlike the MTASS system, this simulator does not need a mission-specific subsystem to handle telemetry; the telemetry formats can be set by run-time parameters. When this strategy is used appropriately, the reuse levels approach 90 percent verbatim code reuse, with the remaining part undergoing minor modifications.

While this 90-percent reuse level has helped reduce software costs and shorten development schedules, it has only done so on a limited class of systems. When the telemetry simulator was reused for a new class of systems (spin-stabilized spacecraft), the system complexity increased, reuse decreased, and run-time performance suffered. MTASS had a similar problem when it was applied to a spacecraft that did not have a sensor on which the original MTASS design depended.

In addition to variations between spacecraft, simulators and ground systems contain many common models. However, the current practice is to create separate systems from separate specifications. The way to account for variations between satellites and to exploit commonality between software systems is to perform domain analysis, rather than attempting to generalize the specification of a single satellite's simulator and ground-support system.

In the FDD, this domain analysis is being done as part of a generalized system development initiative. The attempt to develop generalized software to support multiple flight dynamics applications was based on the experiences of the projects described above. The multimission simulators demonstrated the feasibility of generic architectures, and it had been demonstrated that applying the object-oriented concepts of abstraction and encapsulation was sufficient to increase reuse dramatically. Finally, the existing designs were highly reusable, but had severe limitations in the areas of adaptability and run-time efficiency.

The key concepts selected for generalized system development in the FDD are to perform object-oriented domain analysis, and to have a standard implementation approach for the generalized models. Figure 5 shows a typical diagram from the generalized specifications.

The boxes are generalized superclasses with their subclasses listed inside; Gyro, Sun Sensor, and Star

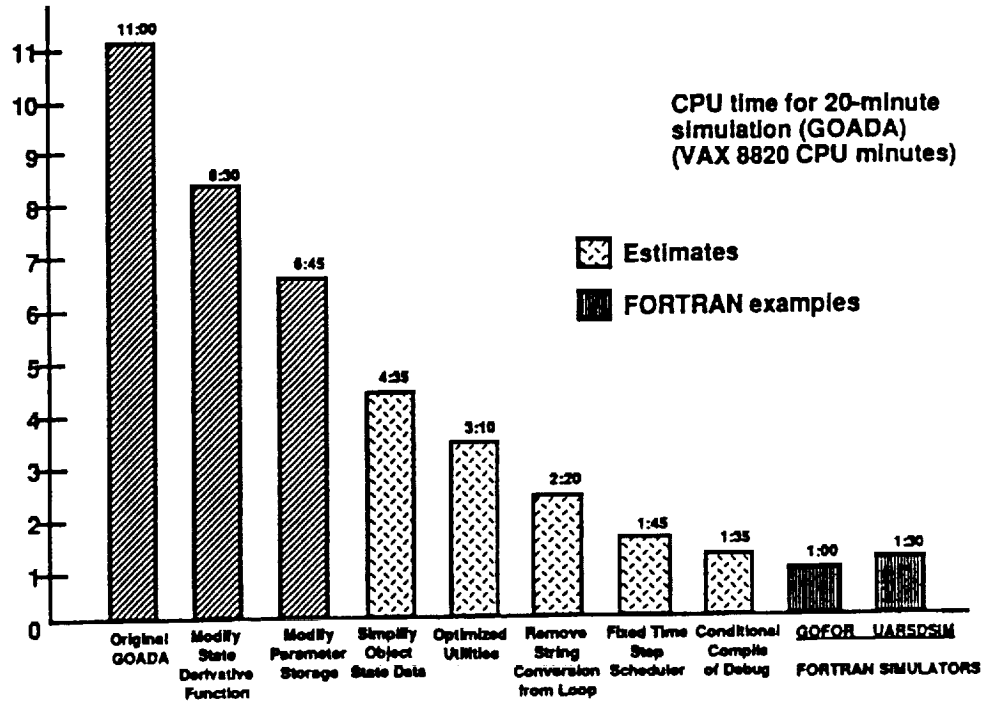


Figure 3. Impact of Performance Goals

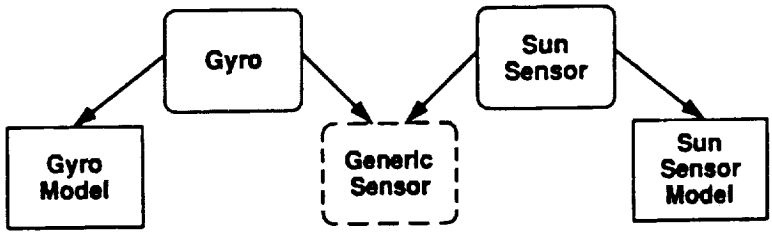


Figure 4. Multimission Telemetry Simulator Design

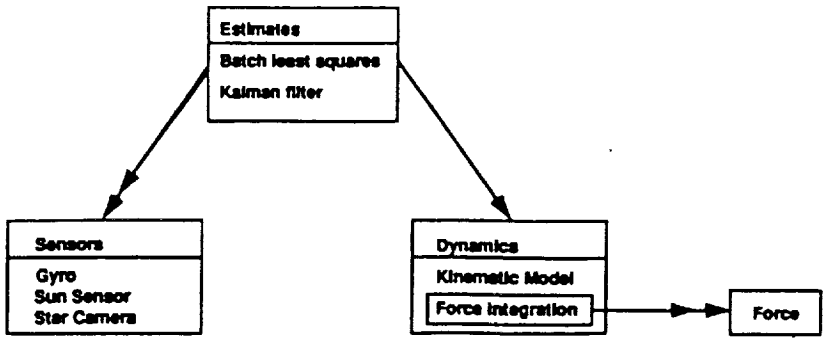


Figure 5. Generalized System Specifications

Camera, for example, are subclasses of Sensor. The arrows between categories represent dependencies between classes. For example, estimators depend on Sensor for measurements and Dynamics for state propagation. These dependencies are matched in the implementation with Ada generic formal parameters. The classes themselves are implemented as abstract data types in Ada packages. Each class shown on the diagram has a corresponding text specification that defines the member functions, user parameters, state data, and dependencies on other classes and categories. Categories also have text specifications for an abstract interface containing the functions common to all classes in the category. With this generalized development effort, object-oriented domain analysis and standard implementation, as well as other features of the object-oriented paradigm, are now being applied to the entire software life cycle.

With the successive generations of object-oriented development efforts defined, the next step is to examine how the SEL's approach has changed between 1985 and 1992. The approach has evolved in what concepts are used, when they are used in the life cycle, and how they are taught.

The concepts of data abstraction and encapsulation, used from the beginning, have themselves enabled the high reuse observed on the MTASS system; even the second Ada simulator attained higher reuse than is typical for similar FORTRAN simulators. The multimission telemetry simulator introduced the idea of inheritance by taking a general model for sensors and tailoring this model for each type of sensor. It also introduced the idea of parameterizing dependencies with Ada generic formal parameters. The generalized application work added the use of abstract data types, where previous systems had implemented objects as state machines. The generalized systems also have a superclass/subclass hierarchy limited to superclasses (called "Categories") and one level of subclasses for each superclass. Dynamic binding is coded using Ada case statements, not an object-oriented programming language feature.

Having support for object-oriented programming in Ada would remove the need to write this code, which would reduce development costs. However, the simple data abstractions provided by Ada packages have already increased reuse levels from approximately 40 to approximately 90 per cent of the delivered code, so the remaining potential cost reductions are dominated by those already attained. Dynamic binding would reduce the tedium of implementing case statements to handle run-time dispatching, but it is not the most important characteristic of object-oriented programming languages from a project cost point of view.

In addition to the increased reuse, the evolution to object-oriented development affected the reliability and changeability of the system. Table 1 shows the effort needed to determine what change is necessary to correct an error or to otherwise enhance a system.

Table 1. Changes Needed to Correct Errors or Enhance System

Effort to Isolate Changes					
Project	< 1 hr	1 hr-1 day	1 day-3 day	> 3 day	Total
GOESIM	116	102	27	7	262
UARSTELS	205	77	10	5	297
SAMPEXTS	8	7	0	0	15

These data are shown for three telemetry simulators. GOESIM is an early Ada project whose design is similar to previous FORTRAN projects. UARSTELS is the first simulator in the multimission telemetry simulator family, and SAMPEXTS is a simulator that reuses from UARSTELS. The second-generation systems have a far greater proportion of changes that take less than one hour to isolate. These results support the claim that object-oriented designs produce systems that are more easily modified because of the information hiding provided by objects and classes.

The types of errors that occur also changed over time. Table 2 shows the classification of errors for the same three systems described above.

These data show that the development of UARSTELS, the initial second-generation system, was slightly more error prone than other projects. While overall errors were increasing, though, errors relating to interfaces and data structures were substantially reduced. Again, this is consistent with the perceived benefits of abstraction and information hiding. Even more striking is the complete elimination of interface errors for high-reuse projects such as SAMPEXTS.

The other notable change is in how OOT affected the development process. In the MTASS system, it had minimal impact, as the design approach was structured, with the object orientation being recognized during coding. Both generations of simulators used object-oriented design and object-based coding based on Ada packages; the generalized system project added an object-oriented approach to defining specifications. It is anticipated that having an object-oriented view throughout the life cycle will make the use of the technology easier by removing the need to recast functional specifications into an object-oriented design.

While object-oriented analysis has not been used for most systems, the high-reuse architectures have been influenced by how the specifications are written. Typical specifications have focused on a single satellite mission, and they specify the simulation and ground-support software separately. The building of the high-reuse MTASS and telemetry simulator systems was possible because the flight dynamics analysts wrote a single specification for the UARS and EUVE missions; the simulator and ground-support systems were still specified separately. The limitations of these specifications is one factor that led to a domain-analysis approach, so that a wider range of satellites can be supported and commonality between ground support and simulation can be exploited. The domain-analysis team switched from a structured to an object-oriented approach as they attempted to write a generalized specification.

Because the generalized system development is still in design, the impact of object-oriented analysis cannot yet be measured. But the use of object-oriented design has changed the development process by shifting work to the design phase. This is due to the high reuse allowing the production of an initial build by integrating existing components. SAMPEXTS thus demonstrated a system that met a large proportion of the requirements at the Critical Design Review. Table 3 shows the distribution of developer effort over the main phases of a development project.

Table 2. Classification of Errors

	Error Class						
	Data Startup	Computational	Logic	External Interface	Internal Interface	Initialization	Total
GOESIM	52	21	10	10	13	21	127
UARSTELS	25	40	43	9	3	39	153
SAMPEXTS	0	4	3	0	0	3	10

Table 3. Developer Effort Over Main Development Phases

Effort Distribution by Phase			
Project	Design	Code	Test
GOESIM	29%	44%	27%
UARSTELS	25%	39%	36%
SAMPEXTS	48%	18%	34%

The SEL provided training in Ada and design techniques for the early Ada simulator experiments, but not for the later multimission simulators. The MTASS FORTRAN system involved no training in OOT, as the project did not set out to use a new language or design technology. The subjective experience of the SEL has been that the application of OOT was not so intuitive as expected, as functional decomposition has been successfully applied for more than 15 years. The SEL, recognizing that transition to a new technology must factor in the time required to learn the new way of thinking, is creating a new training program that captures the lessons learned on previous projects and describes the overall object-oriented software development process as well as specific language and design concepts.

The goal of bringing new technology into the SEL is to measurably improve the software development process. Figure 6 shows the project characteristics of the three multimission simulator projects.

The UARSTELS project was developed to be reused for future simulators, and the projects labeled EUVETELS and SAMPEXTS represent the first two projects to reuse this architecture. Costs were reduced by a factor of 3, change and error rates were reduced by a factor of 10, and project cycle time was cut roughly in half. However, we have already shown that when an attempt was made to reuse this architecture for a different class of projects there were difficulties adapting the code, and run-time performance was unsatisfactory.

The generalized system effort is attempting to gain the benefits shown for this single family of projects over a wider variety of flight dynamics applications. This will allow the FDD to support more missions simultaneously, and it will free resources to concentrate on improving existing capabilities or defining new ones.

SOME CONCLUSIONS

This paper addresses the question, "Is Object-Oriented Technology, then, truly the most influential method studied by the SEL to date?" The conclusion of the SEL is that OOT does promote reuse, sometimes even neglecting other important issues like run-time efficiency. When coupled with domain analysis, OOT enables high reuse across a range of applications in a given environment. While the reuse expectations were met, the use of OOT was not so intuitive as expected, partly because the technique was new to an organization with a mature structured development process. The other factor affecting the ease of transition is the inherent and growing complexity of flight-dynamics problems; OOT may be a better process but, in addition to software techniques, skilled designers are still needed to solve difficult problems.

Still, few (if any) of the other technologies studied here have effects so widespread or so profound as OOT. In fact, OOT is the first technology that covers the entire development life cycle in the FDD. It is an entirely new problem-solving paradigm, not simply a new way of performing familiar tasks in a traditional life cycle. It has been demonstrated to

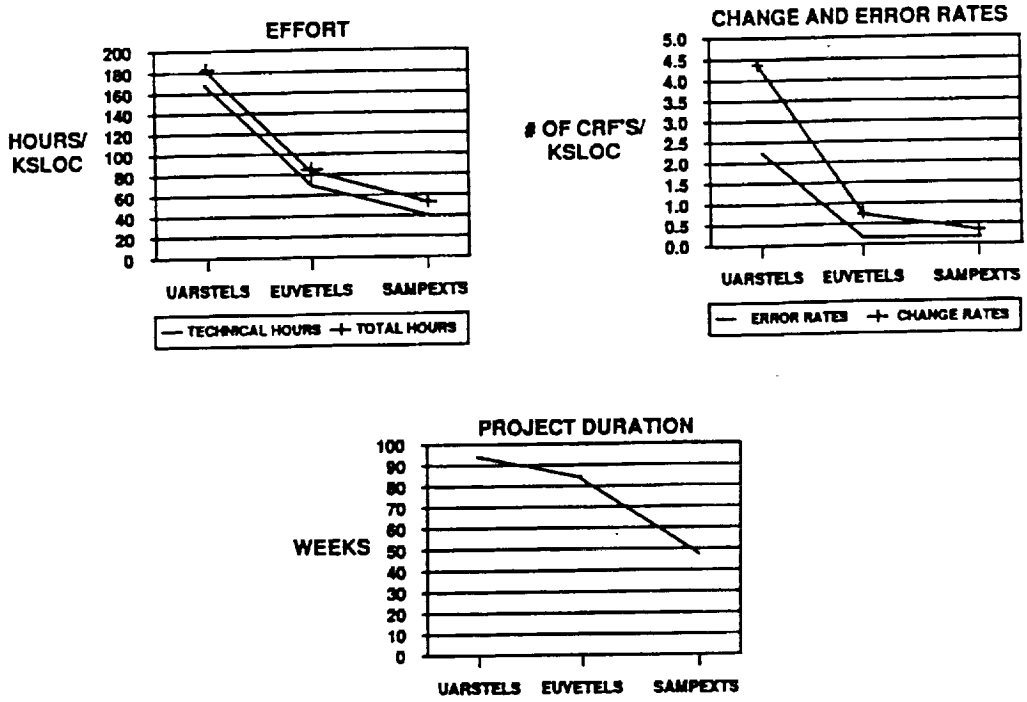


Figure 6. Project Characteristics, Multimission Simulators

expand the reusability and reconfigurability of software, with resultant improvements in productivity and development cycle time. In this sense, OOT is arguably the most influential technology studied by the SEL.

REFERENCES

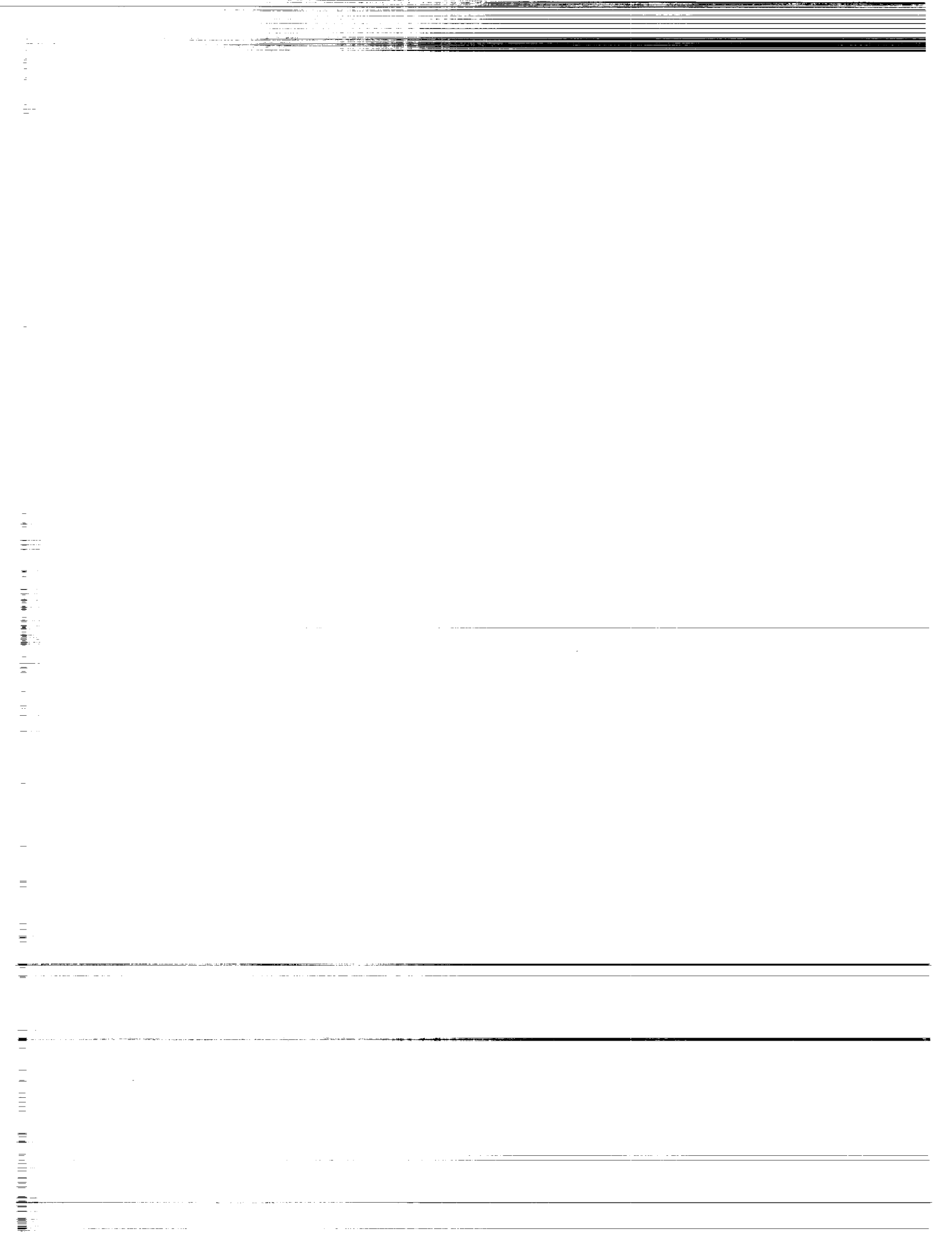
1. McGarry, Frank E., and Waligora, Sharon, "Recent Experiments in the SEL," *Proceedings of the Sixteenth Annual Software Engineering Workshop*, Greenbelt, MD, December 1991, pp. 77-85.

2. Basili, Victor R., and Katz, Elizabeth E., "Software Development in Ada," *Proceedings of the Ninth Annual Software Engineering Workshop*, Greenbelt, MD, November 1984, pp. 65-85.

3. Booch, Grady, *Software Engineering With Ada* (First Edition), Benjamin/Cummings, Menlo Park, CA, 1983.

4. Seidewitz, E., and Stark, M., *General Object-Oriented Software Development*, SEL-86-002, August 1986.

STANDARD BIBLIOGRAPHY OF SEL LITERATURE



STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop*, August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop*, September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop*, September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study*, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment*, T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3)*, W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment*, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop*, November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation*, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model*, A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop*, November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*, J. F. Cook and F. E. McGarry, December 1980

10000229
1201/1700

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development (Revision 3)*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1206, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1993

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

10000229
1201/1700

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-006, *Monitoring Software Development Through Dynamic Variables*, C. W. Doerflinger, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

10000229
1201/1700

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/ Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/ Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

10000229
1201/1700

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-201, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 2)*, L. Morusiewicz, J. Bristow, et al., October 1992

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

10000229
1201/1700

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[9]Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology*, January 1992

10000229
1201/1700

[10]Basili, V., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

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

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

C-2

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, TR-3048, University of Maryland, Technical Report, March 1993

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

10000229
1201/1700

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS '92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

[0]This document superseded by revised document.

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10]This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11]This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

10000229
1201/1700