N94-35436

2

P- 11

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.

PRECEDING PAGE BLANK NOT FILMED PAGE 2-34 INTENTIONALLY BLANK

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:

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

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 $\in$ [69%,100%] OR LUIPAR $\in$ [75%,100%]
ΔH = 0.32, PH=0.82,#PV=39

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

LUCALLS $\in$ [66%, 100%]
ΔH = 0.30, PH=0.81,#PV=42
    LUISUBP $\in$ [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 $\in$ [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 $\in$ [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 $\in$ [58%, 100%]
ΔH = 0.13, PH=0.71,#PV=62

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

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

LUINST $\in$ [76%, 100%]
ΔH = 0.22, PH=0.77,#PV=35
    LUVCUDEC $\in$ [45% ,100%]
    OR LUVCTOT $\in$ [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 $\in$ [77%, 100%]
ΔH = 0.09, PH=0.67,#PV=49
    LUCUDEC $\in$ [48% ,100%]
    ΔH = 0.37, PH=0.84,#PV=25

LUCUDEC $\in$[60%,100%] OR LUCTOT $\in$[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 $\in$ [65%, 100%] OR LUSUBP $\in$ [65%, 100%]
ΔH = 0.09, PH=0.67,#PV=52

LUOBJ $\in$ [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 $\in$ [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 $\in$ [76%, 100%]
ΔH = 0.22, PH=0.77,#PV=35

LUCUDEC $\in$ [60%,100%] OR LUCTOT $\in$ [58%,100%]
ΔH = 0.09, PH=0.68,#PV=62
  LUINST$\in$ [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 $\in$ [0%, 57%] OR LUCALLS $\in$ [0%, 56%]
ΔH = 0.04, PH=0.38,#PV=92
  LUPARM$\in$ [66%,100%] OR LUPARPV $\in$ [72%, 100%] OR LUPARPD $\in$ [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.