International Journal of Software Engineering
and Knowledge Engineering
Vol. 13, No. 4 (2003) 395–417
© World Scientific Publishing Company

TOWARDS INDUSTRIALLY RELEVANT FAULT-PRONENESS MODELS*

GIOVANNI DENARO[†] and MAURO PEZZÈ[‡]
Università degli Studi di Milano-Bicocca,
via Bicocca degli Arcimboldi 8, Milano, I-20126, Italy
[†]*denaro@disco.unimib.it*
[‡]*pezze@disco.unimib.it*

SANDRO MORASCA
Università degli Studi dell'Insubria,
via Valleggio 11, Como, I-22100, Italy
Sandro.Morasca@uninsubria.it

Estimating software fault-proneness early, i.e., predicting the probability of software modules to be faulty, can help in reducing costs and increasing effectiveness of software analysis and testing. The many available static metrics provide important information, but none of them can be deterministically related to software fault-proneness. Fault-proneness models seem to be an interesting alternative, but the work on these is still biased by lack of experimental validation.

This paper discusses barriers and problems in using software fault-proneness in industrial environments, proposes a method for building software fault-proneness models based on logistic regression and cross-validation that meets industrial needs, and provides some experimental evidence of the validity of the proposed approach.

Keywords: Software process; software testing; software metrics; software faultiness; fault-proneness models; logistic regression; cross-validation.

1. Introduction

Assuring an acceptable level of quality of software at realistic costs is a crucial issue for software development. The considerable improvement of software development technology and tools has greatly increased productivity and quality, but software products are far from being perfect, and analysis and testing still account for a large portion of the development costs.

A key factor for monitoring and controlling the quality of software and the effectiveness and costs of analysis and testing is the ability of both measuring the amount of faults found with testing (*software faultiness*) and predicting the distribution of faults before testing (*software fault-proneness*). Software faultiness is

*This paper is an extended version of [4].

relatively easy to measure and helps tune the software development process, but it is of little help in allocating resources and anticipating costs and problems of analysis and testing. On the contrary, software fault-proneness cannot be directly assessed before testing and cannot be easily estimated, but it is extremely helpful in anticipating analysis and testing problems, and in planning effort [6].

A lot of work has been devoted to identifying directly measurable software attributes that allow the estimation of software fault-proneness, i.e., the probability of presence of faults in the software. A number of software characteristics can be captured with static software metrics, e.g., number of lines of code, cyclomatic complexity [21], number of operators and operands [10], number of knots [33]. Although none of the metrics introduced so far is deterministically related to software fault-proneness, the relationships between software fault-proneness and suitable subsets of software metrics have been empirically proved by many authors (see, for instance, [9, 1, 12, 20, 15, 30, 19, 8]).

Unfortunately, the models proposed so far for relating software fault-proneness with subsets of static software metrics apply to specific cases only, but have not been proved relevant for a general class of programs. The search for the "perfect" static metric, i.e., a metric directly related to fault-proneness, and for the "general" fault-proneness model, i.e., a model valid for all or a large class of software, has not produced practical results so far. On the contrary, many scientists have been successful in building prediction models based on sets of metrics tailored to specific application domains or processes. The search for specific fault-proneness models investigated many different approaches: methods based on machine learning principles such as decision trees [31, 28] and neural networks [17], probabilistic approaches such as Bayesian Belief Networks [6], statistical techniques such as discriminant analysis [25] and regression [14], or mixed techniques such as optimized set reduction [2] and the combined use of rough set analysis and logistic regression [24].

Different approaches provide different insight into the problem of deriving suitable fault-proneness models. Some investigate the existence of valid models, others demonstrate the validity of a particular technique for building fault-proneness models, yet others provide different fault-proneness measures. Not all of them are industrially applicable; few of them have been throughly experimentally validated; to the best of our knowledge, none of them has been extensively applied in an industrial setting yet.

In this paper, we investigate the industrial relevance of fault-proneness models. We first describe the main characteristics that make a fault-proneness model suitable for industrial applications, i.e.,

- the model can be built and validated with a relatively small amount of historical data,
- the model produces a continuous fault-proneness indicator thus allowing modules to be ordered according to their fault-proneness,
- there is empirical evidence of the scope of validity of the model.

We then introduce an approach for producing industrially relevant fault-proneness models, using logistic regression [11] and cross-validation [32]. Cross-validation allows models to be validated with a relatively small amount of historical data, while logistic regression produces models that give a continuous fault-proneness indicator.

We finally identify a class of software products within which a fault-proneness model should be valid and we present empirical data that illustrate the validity of the model for an instance of the identified class.

Previous work on logistic regression (e.g., [14, 24]) uses data splitting for selecting a valid model and provides only preliminary empirical evidence of the validity of the approach. Data splitting requires a large amount of historical data and is thus rarely applicable in industrial software engineering settings.

This paper is organized as follows. Section 2 discusses the characteristics of industrially relevant fault-proneness models, proposes logistic regression and cross-validation to build industrially relevant fault-proneness models and identifies possible scopes of validity of such models. Section 3 presents an empirical evaluation of the proposed fault-proneness models and discusses the costs of introducing a fault-proneness model in an industrial setting. Section 4 briefly overviews the main approaches to software fault-proneness indicating important results, limitations and relation with the work presented in this paper. Section 5 summarizes the results presented in this paper and describes ongoing work.

2. Software Fault-Proneness

This section describes the results of our research for industrially relevant fault-proneness models. First, we discuss the process of building and using fault-proneness models, highlighting problems of construction and usage in industrial settings. We then describe our technique for building fault-proneness models based on logistic regression and cross-validation. This technique meets the main requirements for industrial applicability. Finally, we present the problem of identifying the range of validity of the built models. It is very unlikely that there exist fault-proneness models with general validity, i.e., models that can accurately predict the faultiness of software modules of every application. However, researchers agree on the existence of fault-proneness models valid within specific classes of products [16, 7]. We outline the common characteristics of a relevant class of industrial applications that are expected to share fault-proneness models.

2.1. *A software fault-proneness process*

The use of fault-proneness models comprises a *construction* and a *usage* phase. The construction phase is required since there are no generally valid models and thus specific models must be constructed for the different application domains. The construction phase includes some key steps:

- Identification of the target domain: models are valid only within specific classes of applications. Using fault-proneness models for sets of programs that do not meet these requirements requires special care, especially for the validation and tuning phases.

- Analysis of historical data: fault-proneness models are built starting from faultiness data of past applications of the target class of products. The completeness of the available data must be accurately checked.
- Construction of fault-proneness models: this step consists of the construction of a large number of models using statistical tools.
- Selection of a significant model: the best model is selected using methods for validating the quality of the constructed models.

The usage phase includes three main steps:

- Use of the selected model for predicting the fault-proneness of new software products before testing.
- Validation of the results during the testing phase: the faultiness data can help validate the efficacy of the model for the new product and reveal possible discrepancies.
- Tuning of the model at the end of the testing phase: the evolution of the software team and the introduction of new methodologies may change the validity of the model. Fault-proneness models must be recreated periodically to tune the models, increase their precision with the aid of additional data, and keep them valid with respect to the evolution of the software development environment.

The industrial applicability of fault-proneness models depends on the construction of fault-proneness models and the validity of the constructed models. Many fault-proneness models produce discrete indicators of fault-proneness, while others produce continuous indicators. Discrete indicators allow software modules to be grouped in classes according to the estimated fault-proneness. Discrete classifications are often too coarse to be flexible enough for practical use in industrial environments. On the contrary, continuous indicators allow software modules to be sorted according to their fault-proneness. Such orders allow for a much finer allocation of testing time and resources and are thus preferable, if not required, for many industrial applications.

Suitable models can be selected with many techniques, which require different amount of historical data. Techniques applicable with a relatively small amount of historical data are more likely to be applicable in industrial environments, where the sets of historical data are seldom very large.

The potential negative impact of the use of wrong fault-proneness models in industrial settings calls for some experimental evidence on the validity of fault-proneness models at least for some relevant classes of software products.

2.2. *Logistic regression and cross-validation*

We propose to use (1) logistic regression for building fault-proneness models based on historical data and (2) cross-validation for evaluating the models according to their ability of predicting fault-proneness of new software. Models built by means of logistic regression provide a continuous fault-proneness indicator. We have already commented that continuous indicators allow for planning much more flexible allocations of the testing resources, e.g., focusing the testing on given percentages of the most fault-prone modules first. This satisfies a key requirement of industrial software engineering settings. In this respect, logistic regression is more useful than other modeling techniques, e.g., classification trees, which provide discrete classifications. Cross-validation validates the constructed models by reusing the same historical data used in the building procedure. This makes our technique applicable even with relatively little amount of data, meeting the second important requirement of industrial software engineering settings. In this respect, cross-validation outperforms other evaluation techniques (namely, data splitting techniques) that require a dedicated part of the data for testing the models and then need bigger amount of data for achieving statistically significant results.

Logistic regression and cross-validations are presented in detail in [11] and [32]. Here we recall only the elements required to make the paper self-contained.

A logistic regression model estimates the probability that an object belongs to a specific class, based on the values of some quantified attributes of the object. The variable that describes the classes to be estimated is called dependent variable of the model. The variables that quantify the object attributes are called explanatory (or independent) variables of the model. In our study, we instantiate logistic regression as follows:

- objects to be classified are software modules;
- classes that classify modules are: faulty and non-faulty;
- dependent variable of the models (denoted as Y) assumes value 1 for faulty modules and 0 for non-faulty ones;
- explanatory variables are software metrics that quantify the attributes of the software modules.

Technically, a logistic regression model having Y as dependent variable estimates the probability of the event $[Y = 1]$ with the following equations:

$$p(Y=1) = \pi(Linear) = \frac{e^{Linear}}{1+e^{Linear}}$$

$$Linear = C_0 + C_1X_1 + C_2X_2 + \ldots + C_nX_n$$

where X_1, X_2, $\ldots, X_n$ are n explanatory variables and the coefficients C_0, C_1, $\ldots, C_n$ identify a linear combination ($Linear$) of them.

Logistic regression does not assume any strict functional form to link the explanatory variables and the probability function. Instead, this functional correspondence has a flexible S-shape that can adjust to several different situations. The coefficients $C_0, C_1, C_2, \ldots, C_n$, are estimated by maximizing the likelihood function on the set of available observations, which are assumed to be independent. In our experiments, the observations are vectors that contain the metrics and the value of Y for software modules. For each traced module, the value of Y is 1 if at least one fault is known and 0 otherwise. The metrics were directly measured by means of commercial and prototype tools. We rely on the approximation that the subject application is stable enough to have no faults except the known ones.

A logistic regression model is preliminarily assessed based on the following statistics:

- p_i, the statistical significance of coefficient C_i, which provides the probability of obtaining the estimated value of C_i or a more extreme one. In other words, p_i measures the probability that we erroneously believe that the metric X_i has an impact on the estimated probability that a module is faulty. A significance threshold of 0.05 is commonly used to assess whether an explanatory variable is a significant predictor;
- R^2, the goodness of fit, which provides the percentage of uncertainty that is attributed to the model fit. This coefficient is not to be confused with least-square regression R^2: they are built upon very different formulae, even though they both range between 0 and 1. The higher the R^2, the higher the effect of the explanatory variables, the more accurate the model. As opposed to the R^2 of least-square regression, high values of R^2 are rare for logistic regression. Values of R^2 greater than 0.3 can be considered as good in the context of logistic regression.

Our technique for deriving models of software fault-proneness applies logistic regression to historical datasets in an explorative fashion. We compute many logistic regression models with different sets of the available metrics as explanatory variables. We start by computing all the possible models having one single metric as explanatory variable. Then, we compute all the models based on two metrics and so on combinatorially. For example, in our experiments we have built up to 1,300,000 models for a single application. We use the R^2 and p_i statistics for a preliminary screening of the constructed models, but the number of statistically significant models is generally high, up to 2,300 in our experiments.

We cannot imagine a scenario in which it makes sense to use 2,300 models (which might provide discordant predictions) for tuning the testing process. Thus we need to select the model with the best quality of prediction.

Cross-validation assesses the quality of prediction of a logistic regression model referring to the same data used to build the model, and thus produces statistically valid indicators of quality also for relatively small amount of data. Given a set of explanatory variables, the cross-validation of the corresponding logistic regression model consists of the following steps:

- Partition the observations into n approximately equal-size sets;
- Compute n logistic regression models for the given explanatory variables. Each

model is computed based on the observations belonging to $n-1$ sets of the partition, while the observations belonging to the remaining set are used as test-set;

- Cumulate the results achieved for the n test sets in order to measure the success rate in classifying faulty and non-faulty modules of the entire initial data set;
- Iterate the procedure with different partitions a number of times, and average the results.

In common practice, the observations are partitioned into 10 sets and the partitions are stratified, i.e., in each partition the distribution of the objects belonging to each class is approximately the same as in the whole dataset (*stratified ten-fold cross-validation*). Stratified ten-fold cross-validation is used in this work for evaluating the quality of prediction provided by logistic regression models.

Several different indicators may be used for measuring the success rate of a model. Depending on the indicator chosen, different models may be selected. In our empirical studies, we have evaluated three different strategies, which we describe next:

Best overall completeness. Overall completeness is defined as the proportion of modules that are classified correctly:

$$\text{overall completeness} = \frac{\text{CPFM} + \text{CPNFM}}{\text{M}}$$

where, $CPFM$ is the number of correctly predicted faulty modules, $CPNFM$ is the number of correctly predicted non-faulty modules and M is the total number of modules.

This parameter measures the ability of models to correctly classify the target software modules, regardless of the category in which the modules have been classified.

Best faulty module completeness. Faulty module completeness is defined as the proportion of faulty modules that are correctly classified as faulty by the model:

$$\text{faulty module completeness} = \frac{\text{CPFM}}{\text{FM}}$$

where, $CPFM$ is the number of correctly predicted faulty modules and FM is the total number of actual faulty modules.

This parameter measures the ability of the models to identify the most fault-prone part of the target software.

Best faulty module correctness. Faulty module correctness is defined as the proportion of faulty modules within the set of modules that are classified as faulty by the model.

$$\text{faulty module correctness} = \frac{\text{CPFM}}{\text{PFM}}$$

where, $CPFM$ is the number of correctly predicted faulty modules and PFM is the total number of predicted faulty modules.

This parameter measures the efficiency of a model, in terms of the percentage of actual faulty modules among the ones that are candidates for further verification.

The second and third of such parameters focus on evaluating the performance of models with respect to the identification of faulty modules. We agree with other researchers that faulty modules can be considered more relevant than non-faulty ones as far as verification issues are concerned ([24]). Early identification of faulty modules is likely to increase the effectiveness of the testing process and improve the overall quality of the delivered software. However, we have not been able in our experiments to identify relevant differences between such three selection strategies.

2.3. *Scope of fault-proneness models*

The evaluation of the quality of prediction is necessary but not sufficient. Models that show good prediction quality may be good for some applications, but bad for others. Thus, we need to identify the range of validity of a type of fault-proneness models, i.e., the set of software applications for which the models are expected to provide meaningful results.

Figure 1 illustrates the issue. Fault-proneness models that are built within a *class of software product* may not be valid for products outside the class. In this

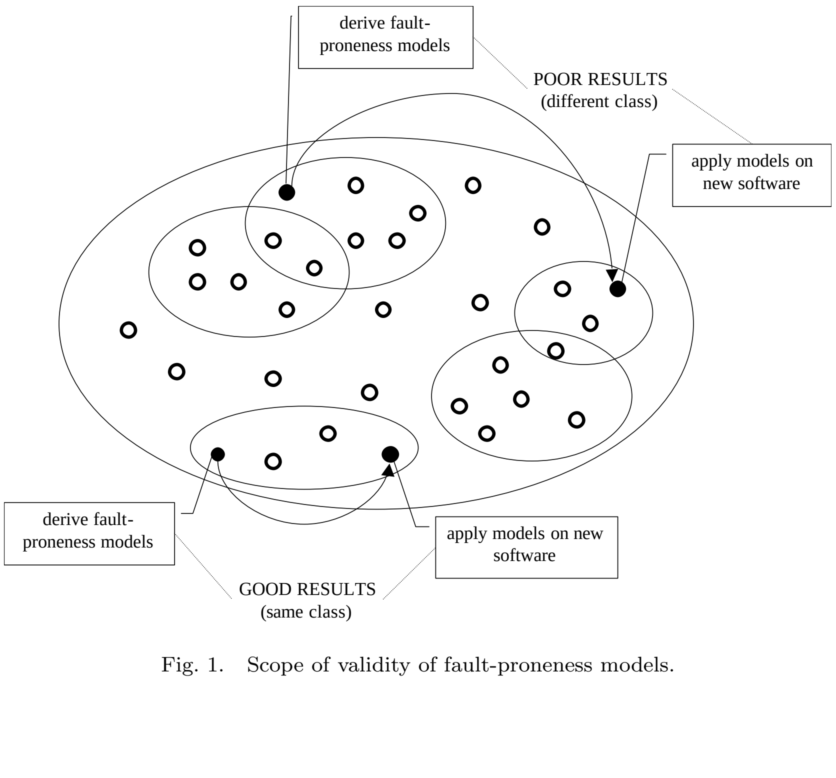
derive fault-proneness models
POOR RESULTS (different class)
apply models on new software
derive fault-proneness models
apply models on new software
GOOD RESULTS (same class)

Fig. 1. Scope of validity of fault-proneness models.

case, they may lead to poor results. On the contrary, using fault-proneness models for products of the same class is much more likely to provide reliable results. Thus, it is important to identify relevant classes of products.

In our work, we focused on classes of software products that:

- belong to the same application domain, i.e., solve similar problems;
- are developed with similar methods, techniques, processes, and implementation environments;
- are developed by homogeneous teams, i.e., teams of people with similar technical background and working with similar teamwork procedures.

Such classes correspond to many industrially relevant cases, where software is developed for the same application domains by teams working with fairly stable development environments. Such classes are expected to share faultiness characteristics. In fact, it is common practice to plan test and analysis activities based on the experience in the development groups and the application domains. Common instances of such classes are families of products, i.e., software products developed over time by the same company. In this case, the products solve a similar problem, if not even the same one; they are usually developed with similar methods, techniques, process, and development environment; and they are built by people who are usually hired with similar criteria and requirements. Other works present preliminary evidence that software complexity and fault-proneness can be stable across products of the same organization [16, 7].

3. Empirical Assessment

We empirically evaluated the relevance and applicability of the technique presented in this paper. In particular we designed experiments to check for:

1. The possibility of building fault-proneness models. Although several papers propose fault-proneness models, there is still a limited amount of experimental data available in the scientific literature. The first, although not the main, goal of our empirical studies is to provide additional data to increase the confidence in the existence of such models.
2. The suitability of cross-validation for predicting the relevance of fault-proneness models built with logistic regression. An important goal of our empirical studies is to assess cross-validation for evaluating the relevance of fault-proneness models, since cross-validation allows for the validation of models with a relatively small set of historical data.
3. The validity of fault-proneness models across the classes of software products that belong to the same application domain and are developed with similar methodologies by homogeneous development teams.

3.1. *Assessing our technique for deriving fault-proneness models*

We conducted a first empirical study to validate the first two research hypotheses outlined above, i.e., the possibility of building fault-proneness models and the suitability of cross-validation for predicting the relevance of fault-proneness models. To this end, we analyzed a program derived from an antenna configuration system developed by professional programmers. The availability of detailed data on faults revealed during system testing and operational use makes the program extremely useful for studies on fault-proneness. Since this program has already been used in other scientific experiments ([27, 8, 18]), the data available for it can be trusted as a valid benchmark for empirical studies.

This software system contains over 10,000 lines of code and provides a language-oriented user interface for the configuration of antenna arrays. The users enter high level descriptions and the program computes the corresponding antenna orientations. The program is composed of 136 modules, of which 109 do not contain faults and 27 contain at least one fault. The term *module* is used to refer to a subprogram as in [8, 31, 28]. The total number of documented faults is 39.

For each module in the dataset we measured the 33 software metrics listed in Appendix A. Due to the exploratory nature of this first experiment we did not carry out preselection of metrics, but we used the ones made available by popular commercial and public domain tools.

In the data analysis, we classified modules as either faulty or non-faulty, based on the available data. We associated a variable Y with each module in the dataset. The value of Y is 1 if the module contains at least 1 fault, 0 otherwise. Considering Y as the dependent variable, we ran logistic regression with different sets of software metrics as explanatory variables, and we obtained a number of logistic regression models. For a given software module, these models estimate the probability of Y to be 1, i.e., the probability of the module to be faulty. In particular, we built models for all possible sets of metrics, with cardinality between 1 and 6, selectable among the 33 available software metrics. This yielded more than 1,300,000 models. Since models with too many explanatory variables lead to less credible increments in the model goodness of fit, the threshold of 6 metrics was a good compromise between the number of models to be computed and the potential significance of the models.[a] For the sake of simplicity, in an initial study such as the one documented in our paper, we have used linear logistic regression models, i.e., models in which the exponent of e is a linear function of the explanatory variables. Other models might be taken into consideration, for instance those containing quadratic terms (e.g., X_1^2) or so-called interaction terms (e.g., $X_1 * X_2$). Here, we wanted to ascertain the existence of general trends, rather than identify more precise estimation functions for probabilities.

[a]When the number of explanatory variables reaches the number of observations, the R^2 of a logistic regression model becomes 1 by construction.

Table 1. Summary of the computed fault-proneness models.

Number of expl. variables	Number of models	Best R^2
1	33	0.1735
2	528	0.2692
3	5,456	0.3241
4	40,920	0.4314
5	237,336	0.4884
6	1,107,568	0.5270

Once all the models were automatically computed, only the ones with good statistical significance of the explanatory variables (all p_i's < 0.05) and goodness of fit ($R^2 > 0.3$) have been considered for further analysis. For such models, we evaluated the quality of prediction by means of cross-validation. The fault-proneness threshold for classifying modules as faulty or non-faulty was fixed to 0.5 (50% probability). We applied cross-validation according to the criteria introduced in Sec. 2. Thus, we selected the three models with the best *overall completeness*, *faulty module completeness* and *faulty module correctness*, respectively.

Table 1 contains a summary of the fault-proneness models that have been automatically produced for the examined software. Column 2 indicates the number of models that contain a given number of explanatory variables (column 1). Columns 3 reports the best values of goodness of fit (R^2) for the models in each set. Good values of R^2 (> 0.3) are achieved for models based on more than two explanatory variables.

Since several models with good significance indices exist, we need a strategy for selecting the best candidates for being used on future projects. We investigated selection strategies based on goodness of fit, i.e., the highest R^2, and quality of prediction, i.e., either highest overall completeness, or highest module completeness, or highest faulty module correctness. During the selection, R^2 was used as discriminator whenever the applied strategy selected more than one model. The four selected models are shown in Appendix A.

Figure 2 plots the performance indices of the four selected models against the selection strategy that was applied. The figure compares the values of R^2, overall completeness, faulty module completeness, and faulty module correctness for the models selected by the four strategies. The diagrams show that:

1. None of the models selected with the different strategies outstands positively or negatively with respect to any other performance indices, and
2. All four selected models present good values for all performance indices.

While high goodness of fit (R^2) is obtained by construction, since we discriminated models according to R^2, the uniformity and quality of the other parameters demonstrate the validity of cross-validation for selecting models. In fact cross-

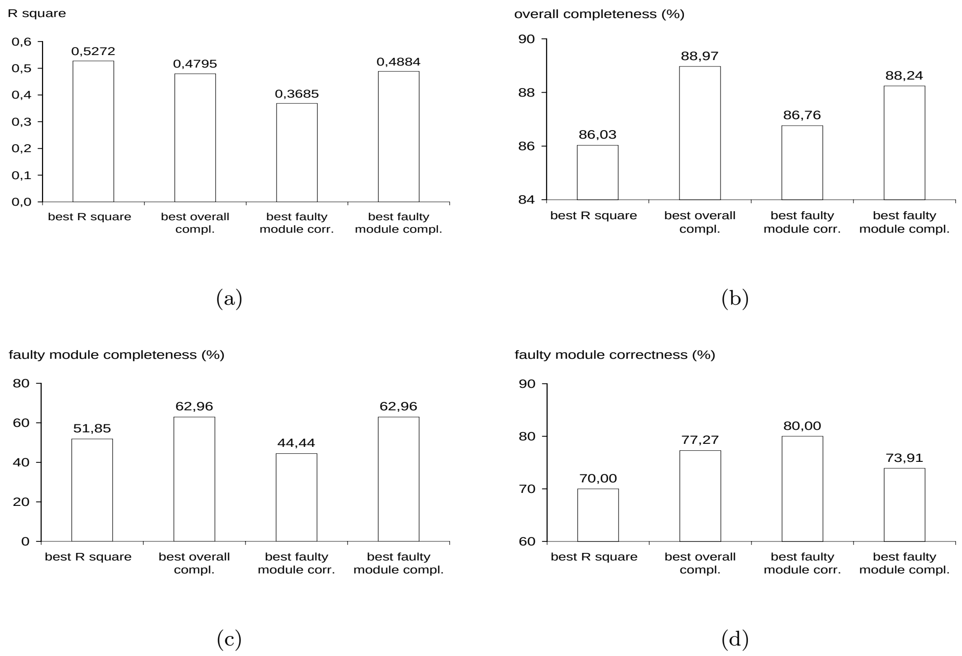

Fig. 2. Parameters of the selected models. The four diagrams report the performance indices for the four selected models. The models are listed on the X axis and the value of one of the four performance indices on the Y axis of each diagram, respectively. In particular, diagram (a) reports the value of R^2, while diagrams (b), (c), and (d) report the overall completeness, the faulty module completeness, and the faulty module correctness.

validation produced models with high prediction indices. This supports the second research hypothesis stated at the beginning of this section.

The usefulness of models that provide a continuous index of fault-proneness is illustrated by using the selected model to sort software modules according to their expected faultiness and then measure the incremental number of faults in the modules. The Alberg diagram ([26]) of Fig. 3 plots the cumulative percentage of faults in the modules of the software under test, ordered according to three different criteria. The solid line represents the cumulative percentage of faults for the program used in this empirical study if its modules are ordered according to the known faults. The dashed and thin lines represent the cumulative percentage of faults if the modules are ordered according to the fault-proneness as provided by two of the selected fault-proneness models. The distance between the optimal order and the order provided by the two models is very small, and thus the models constructed using logistic regression and cross-validation may be successfully used instead of the optimal order to define effective testing strategies. This supports the first research hypothesis stated at the beginning of this section.

Finally, a preliminary interpretation of the models from a software engineering point of view can be given as follows. All four selected models are based on ex-

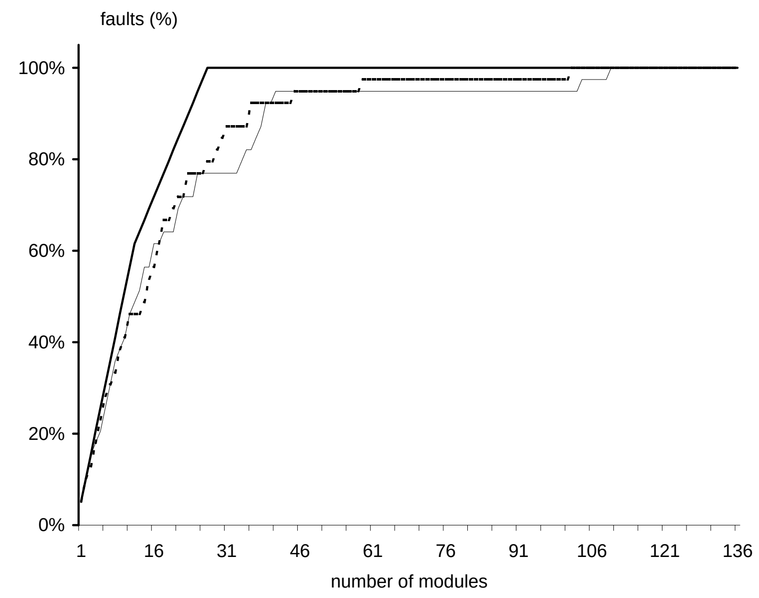

Fig. 3. Cumulative percentage of faults in the antenna configuration modules in different orders. The X axis reports the number of modules of the software system, while the Y axis reports the correspondent cumulative number of faults. In the diagram, the solid line corresponds to the modules ordered according to the known faults; the dashed line corresponds to the modules ordered according to the fault-proneness indicator computed by the best-R^2 model; the thin line corresponds to the modules ordered according to the fault-proneness indicator computed by the best-overall-completeness model.

planatory variables that are representative of the following software characteristics:

Size: several size-related metrics appear in the models, namely eLOC, Lines, and EXEC. Among these, Lines is the one that comprises all components of size. Its coefficient is positive in all models, which confirms the intuitive idea that the likelihood of faults increases with the size of the software code. As for eLOC, its negative sign in the models is due to an interaction with variable Lines, i.e., there is a partial counterbalancing negative effect of eLOC with respect to the positive effect due to Lines. In fact, univariate analysis showed that, in a univariate logistic regression where eLOC is the only explanatory variable, the effect of eLOC is positive.

Comments: all models indicate that the number of comment lines in a module has an impact on the likelihood of faults, since Comments appears to be a statistically significant predictor in all four models. The fact that the coefficient of Comments is negative shows that, in the context of all four models, the higher the number of comment lines the lower the likelihood of faults. We can interpret this variable as an indicator of the pressure during development: a low number of comments may denote high pressure on the programmers, and thus higher probability of introducing errors.

Interface complexity: all models indicate that highly complex interfaces increase fault-proneness. Interface complexity is captured via the metric IC and its two components, i.e., FP, the number of formal parameters, and FR, the number of exit points. At least one metric among IC, FP, and FR is statistically significant in each of the four model. Whenever only one of these measures appears in a model (see the first three models in Table A.1), its coefficient is positive. This confirms the intuition that modules designed for complex interaction patterns, and thus with complex interfaces, tend to be more fault-prone. In the fourth model, both FR and IC are statistically significant: IC has a positive coefficient, while FR has a negative one. Thus, the effect of FR in the model somewhat counterbalances the effect of IC, since there is an obvious interaction between FR and IC.

Internal complexity: all models indicate that the complexity in the code structure, e.g., logic flow complexity (LFC) and the number of input and output nodes (NION), negatively impacts on the presence of faults. This result may sound counterintuitive, since conventional wisdom is that the higher the complexity of a software module, the higher the number of faults it contains. The experimental data reflect the accuracy of design and coding: programmers tend to concentrate more on complex modules than simple ones. Consequently, complex modules are usually designed, coded and tested[b] carefully before being released for integration. Thus usually, system tests and operational use find more faults in simple modules and in module interactions than in complex modules. Our models have been built based on data of faults revealed during system testing or operational use of the application, and the derived fault proneness models confirm these observations.

Readers can find additional data for reproducing the experiments in [4].

3.2. *Assessing the scope of fault-proneness models*

We conducted further experiments to evaluate the third hypothesis stated at the beginning of this section, i.e., to investigate the applicability of fault-proneness models built with our technique within the classes of software products identified in Sec. 2. To this end, we chose the Apache Web servers as target applications and, in particular, we experimented with Apache 1.3 and Apache 2.0. On the one hand, these two applications belong to one of the classes of software applications we are interested in, i.e., they are instances of the class of the industrial applications produced over time by the same company for the same application domain. On the other hand, the open source development process of Apache.org enabled us to access all the information about code and faults, as required to perform the experiments. Fielding et al. described and analyzed the software development process of the Apache servers [23]. They comment that all failures have been automatically

[b]In industrial settings, programmers are usually in charge of unit testing as well.

tracked over time and whenever a failure has been solved the connection between the corresponding fault and faulty code has been tracked 90% of the time. This guarantees that the information we extracted from the historical databases can be widely trusted.

Apache 1.3 and Apache 2.0 well approximate all properties that characterize the identified classes of software:

- They belong to the same application domain, being two versions of the same software product.
- Both are developed with the same open-source development process, which fixes development language, development rule, and synchronization procedures [23]. Developers are free to choose their preferred CASE tools, but most of them rely on similar free software.
- Both are managed by the same core of people who work with a large number of volunteers following precise procedures [23].

The size and complexity of the Apache Web servers are comparable with those of industrial applications, and Apache 1.3 and Apache 2.0 differ from each other for more than 50% of the code, i.e., as for the experiment, they should not be considered a variation of the same application, but two different products developed by a homogeneous team. In many cases, the amount of code reused in industrial software developed by the same company for the same application domain is often larger than 50% of the code.

We collected data from Apache 1.3 and Apache 2.0 and we identified the implementation modules, i.e., the files containing executable statements. For each module we extracted the known faults from the project repositories and we computed a set of software metrics. Here we report the main results of the experiments (the details are described in [5]).

In a preliminary experiment, we used the highest quality models from the antenna configuration system of the previous empirical study to evaluate the fault-proneness of Apache 1.3, and we compared the fault-proneness indicators with the known faults. The results were very poor: the computed fault-proneness does not give more information than a random index. This preliminary experiment demonstrates that the quality of prediction of models is not valid in general. Thus, identifying classes of products within which models are valid is necessary for the industrial applicability of fault-proneness models.

We then applied our technique for building fault-proneness models based on the data from Apache 1.3. With logistic regression we built about 600,000 models based on different sets of metrics. Out of these, we extracted the 2,300 models with acceptable statistical significance and we selected the best models with cross-validation, according to different criteria. The models selected on Apache 1.3 were applied to Apache 2.0 to investigate their usefulness in future projects.

Figure 4 shows an Alberg diagram that compares the performances of the fault-proneness models with the actual faultiness. We can see that the curves correspond-

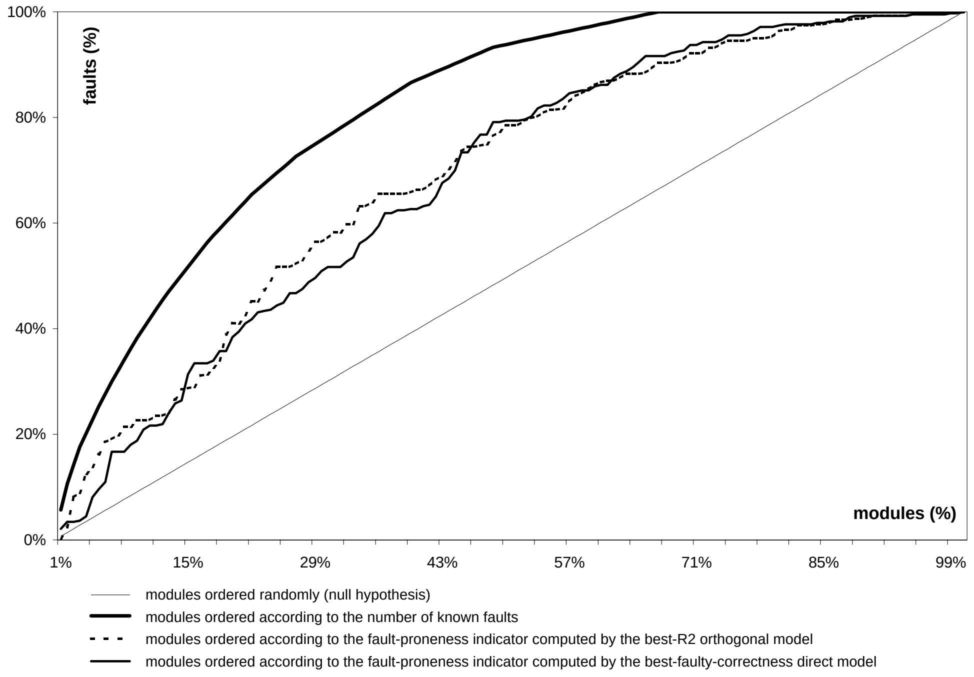

Fig. 4. Cumulative percentage of faults in the Apache 2.0 modules in different orders.

ing to the faultiness estimated with the two fault-proneness models are very close. Moreover, the curves corresponding to the faultiness estimated with the two models tend to approximate the curve corresponding to the actual faults and are significantly different from the curve corresponding to random selection of modules. This confirms that the models derived from Apache 1.3 provide useful predictions for Apache 2.0. As expected, the information provided by the models is not completely accurate. Nonetheless, 50% of the modules that are predicted as most fault-prone by our models are responsible for 80% of the actual faults. Thus, for example, planning a test strategy that focuses on 50% of the most fault-prone modules can significantly improve testing results.

3.3. *Preliminary evaluations of costs*

The last dimension to be considered for industrially applicable techniques is related to costs.

Collecting metrics and applying models for estimating fault-proneness are fairly inexpensive activities if they are supported by automated tools. In our experiment we use commercially available tools that perform these activities in negligible time. The cost of the approach depends mainly on the construction and selection of the models. Such activity requires the construction and cross-validation of a number of models combinatorial in the number of available metrics. With the help of suitable prototype tools (that we opportunely built on top of Java implementations of logistic regression and cross-validation algorithms, as provided in [32]), we have been able to construct and validate the models required in our experiment using a few hours of machine time of a Pentium II PC. Thus, the overall cost is much lower than the cost of testing.

A careful selection of metrics can significantly reduce the costs. In the experiments reported in [5], we also showed that the principal component analysis [13] can be used in conjunction with our technique for optimizing the time requirements. The optimization consists of running principal component analysis on the data set of the collected metrics before applying logistic regression. Principal component analysis allows us to find the orthogonal dimensions of the space of the metrics. In our experiments, we selected 9 orthogonal dimensions that account for 95% of the variance in a space of 38 metrics. We built all the possible models that use the orthogonal dimensions as explanatory variables. Although interpreting these models is more difficult than interpreting models based on direct metrics, their construction required few minutes of machine time. The results of the experiments show that the quality of prediction of these models when used on new software is comparable with the quality of prediction of the models built directly upon primitive metrics.

4. Related Work

Several studies address the relationships between software metrics and software fault-proneness. In the seventies, research tried to identify single metrics that could quantify software complexity ([21, 10]). From the late eighties, more research has tried to relate software complexity to sets of different metrics, deriving so-called multivariate models.

Multivariate models can be built based on the analysis of available data of past projects. Besides the logistic regression used in this paper, other methods have been explored.

Decision trees [29, 31, 28] are able to identify classes of objects based on a set of local decisions. Each node of a decision tree is associated with an explanatory variable. When a node is reached, the actual value of the associated explanatory variables identifies an output edge that leads to another node. During the classification, the tree is traversed, starting from the root node, until a leaf node is reached. Leaf nodes are associated with classification values. Decision trees can be automatically built from sets of historical observations and are particularly useful when the explanatory variables are discrete, although they can be adapted to handle continuous explanatory variable (in this case intervals are considered).

Neural networks [17] are sets of interconnected neurons, each having a number of inputs, an output, and a transformation function. Neural networks are commonly organized in a sequence of layers (at least three), and the neurons of each layer receive as inputs the weighted sum of the outputs of the previous layer. The first layer receives as inputs the explanatory variables and the last layer outputs the resulting classification value. Neural networks natively handle continuous explanatory variables.

Optimized set reduction [2] attempts to determine which subsets of observations from the historical data set provide the best characterizations of the objects to be assessed. Each of the *optimal* subsets is then characterized by a set of predicates (a pattern), which can be applied for classifying new objects. Optimized set reduction can handle either continuous or discrete explanatory variables and provides the expected value of the dependent variable.

All these methods provide simple classifications, i.e., an estimation of which modules are likely to be faulty and which are not. Instead, logistic regression provide a fault-proneness index that can be used for ordering modules.

Logistic regression has been used in empirical software engineering for a number of goals, including estimation of software fault-proneness. It has been used mostly for correlational studies. Here we report on some studies that use data splitting techniques to assess the obtained models. Morasca and Ruhe compare and combine logistic regression with rough set analysis, for identifying the most fault-prone part of a software systems [24]. The empirical study shows that logistic regression and rough set analysis have different strengths and weaknesses and that their combined use can improve the quality of the models. Khoshgoftaar et al. use logistic regression for identifying models for software fault-proneness [14]. The empirical study uses stepwise regression for finding logistic regression models. Stepwise regression is a model generation procedure based on adding new explanatory variables to the model while they significantly contribute to improve the model. Although the paper presents interesting results, stepwise regression may lead to suboptimal models [22]. Moreover, Khoshgoftaar et al. use data splitting to estimate the quality of prediction provided by the models, i.e, part of the data is excluded from the model generation procedure. Since data splitting derives models from a subset of available data, namely it excludes data used to validate the constructed models, it may provide less accurate results for limited amount of available data. Briand et al. [3] use logistic regression to correlate software design metrics with the fault-proneness of the final code. In this paper, we use a combinatorial approach based on building all the models for fixed-size subsets of metrics, and we propose cross-validation to evaluate the constructed models. Cross-validation allows for evaluating the provided quality of prediction without excluding data from the model generation procedure, and thus applies better when limited amount of data are available.

More recent work by Fenton and Ohlsson restates the hypothesis that "software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases" [7]. They support this hypothesis by comparing total fault densities of two consecutive releases of an industrial project, but they recognize that the provided data are not sufficient to validate the hypothesis, and they call for additional experimental data. Our work properly extends the preliminary data presented by Fenton and Ohlsson, since we provide evidence that fault-proneness data computed on a software package are related to faultiness of a different software package with similar characteristics.

Khoshgoftaar and Lanning investigate the stability of principal components of software complexity across different software products [16]. They provide evidence that principal components of software complexity can vary across software products, but they are likely to be stable across products developed by the same organization. Our research goes further: we provide empirical evidence that the stability of the principal components of software complexity for a class of software products allows the use of fault-proneness models across different applications of the same class.

5. Conclusions and Future Work

This paper proposes a new technique, based on the joint use of logistic regression and cross-validation, for modeling the relationships between software metrics and software fault-proneness in the context of industrial software processes. The paper addresses several issues. It provides further empirical evidence of the effectiveness of logistic regression for building models of fault-proneness. It proposes cross-validation to evaluate the produced models in order to obtain accurate results even when a limited amount of data is available. It thus provides an approach that can be effectively applied to many industrial cases where the amount of available data is limited. Finally, it provides initial experimental evidence of the existence of an important set of classes of products that preserve the validity of fault-proneness models built with the approach proposed in this paper. This represents a first important step towards using fault-proneness models within organizational boundaries.

The technique proposed in this paper requires some preliminary work on existing data for building the models of software fault-proneness and tuning the testing process. However, it results in a low overhead on the processes themselves, being easily implemented with automatic elaborations of data that can be automatically collected with commercial tools. In fact, the proposed model selection strategy based on cross-validation enables the automation of the data analysis. The main requirement on the processes for the application of this technique is a mature testing documentation, i.e., detailed description of failures and faults, and their relation with the testing process.

We do not aim at providing a general model of fault-proneness (which might not even exist), but rather a methodology for building different models, for classes of homogeneous products, that can be of practical use if included in the testing processes of groups working on specific product lines. Continuous validation of the current models on the data that become available while progressing in the development is required to limit the divergence of the models from the development practice that may vary for many reasons, including beneficial feedback from the models themselves.

We are now continuing our experiments on new data sets to further validate the methodology for building fault-proneness models and better identify the classes of applications that are expected to share fault-proneness models.

Acknowledgments

We would like to thank Alessandro Orso, Adam Porter, Sergio Scarpati and Maurilio Zuccalà for their important contribution to the experiments. This research has been partially supported by the European Commission within the Best Practice Project PROMOTE and the ESERNET project (IST-2000-28754), and by MIUR (the Italian Ministry of University and Education) in the context of the QUACK project (Cofin Program).

References

1. V. Basili and D. Hutchens, "An empirical study of a syntactic complexity family", *IEEE Trans. on Software Engineering* **9**(6) (1983) 664–672. Special Section on Software Metrics.
2. L. Briand, V. Basili, and W. Thomas, "A pattern recognition approach for software engineering data analysis", *IEEE Trans. on Software Engineering* **18**(11) (1992) 931–942.
3. L. Briand, S. Morasca, and V. Basili, "Defining and validating measures for object-based high-level design", *IEEE Trans. on Software Engineering* **25**(5) (1999) 722–743.
4. G. Denaro, S. Morasca, and M. Pezzè, "Deriving models of software fault-proneness", in *Proc. 14th Int. Conf. on Software Engineering and Knowledge Engineering* (*SEKE-02*), 2002.
5. G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models", in *Proc. 24th Int. Conf. on Software Engineering* (*ICSE-02*), ACM Press, New York, 2002, pp. 241–254.
6. N. Fenton and M. Neil, "A critique of software defect prediction models", *IEEE Trans. on Software Engineering* **25**(5) (1999) 675–689.
7. N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system", *IEEE Trans. on Software Engineering* **26**(8) (2000) 797–814.
8. P. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness", *ACM SIGSOFT Software Engineering Notes* **23**(6) (1998) 153–162, *Proc. ACM SIGSOFT Sixth Int. Symp. on the Foundations of Software Engineering.*
9. G. Gill and C. Kemerer, "Cyclomatic complexity density and software maintenance productivity", *IEEE Trans. on Software Engineering* **17**(12) (1991) 1284–1288.
10. M. Halstead, *Elements of Software Science*, Elsevier, North-Holland, New York, 1977.
11. D. Hosmer and S. Lemeshow, *Applied Logistic Regression*, Wiley-Interscience, 1989.
12. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria", in Bruno Fadini (ed.), *Proc. 16th Int. Conf. on Software Engineering*, IEEE Computer Society Press, Sorrento, Italy, May 1994, pp. 191–200.
13. I. Jolliffe, "Principal component analysis", in *Principal Component Analysis*, Springer Verlag, New York, 1986.
14. T. Khoshgoftaar, E. Allen, R. Halstead, G. Trio, and R. Flass, "Using process history to predict software quality", *Computer* **31**(4) (1998) 66–72.

15. T. Khoshgoftaar, E. Allen, K. Kalaichelvan, and N. Goel, "Early quality prediction: A case study in telecommunications", *IEEE Software* **13**(1) (1996) 65–71.
16. T. Khoshgoftaar and D. Lanning, "Are the principal components of software complexity data stable across software products?" in *Proc. 2nd Int. Software Metrics Symposium*, 1994, pp. 61–72.
17. T. Khoshgoftaar, D. Lanning, and A. Pandya, "A comparative-study of pattern-recognition techniques for quality evaluation of telecommunications software", *IEEE Journal on Selected Areas in Communications* **12**(2) (1994) 279–291.
18. J. M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency", in *Proc. 22th Int. Conf. on Software Engineering*, Limerick, Ireland, June 2000, pp. 126–135.
19. M. Lehman, D. Perry, and J. Ramil, "Implications of evolution metrics on software maintenance", in T. Koshgoftaar and K. Bennett (eds.) *Proc. Int. Conf. on Software Maintenance*, IEEE Computer Society Press, 1998, pp. 208–217.
20. H. F. Li and W. K. Cheung, "An empirical study of software metrics", *IEEE Trans. on Software Engineering* **SE-13**(6) (1987) 697–708.
21. T. McCabe, "A complexity measure", *IEEE Trans. on Software Engineering* **2**(4) (1976) 308–320.
22. P. McCullagh and J. A. Nelder, *Generalized Linear Models*, 2nd edition, Chapman and Hall, London, 1989.
23. A. Mokus, R. Fielding, and J. Herbsleb, "A case study of open source software development: The apache sever", in *Proc. 22nd Int. Conf. on Software Engineering* (*ICSE 2000*), 2000, pp. 263–272.
24. S. Morasca and G. Ruhe, "A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance", *The Journal of Systems and Software* **53**(3) (2000) 225–237.
25. J. Munson and T. Khoshgoftaar, "The detection of fault-prone programs", *IEEE Trans. on Software Engineering* **18**(5) (1992) 423–433.
26. N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches", *IEEE Trans. on Software Engineering* **22**(12) (1996) 886–894.
27. A. Pasquini, A. Crespo, and P. Matrella, "Sensitivity of reliability-growth models to operational profile errors versus testing accuracy [software testing]", *IEEE Transaction on Reliability* **45**(4) (1996) 531–540.
28. A. Porter and R. Selby, "Empirically guided software development using metric-based classification trees", *IEEE Software* **7**(2) (1990) 46–54.
29. J. Quinlan, "Induction of decision trees", *Machine Learning* **1**(1) (1986) 81–106. QUINLAN86.
30. R. Selby and V. Basili, "Analysing error-prone system structure", *IEEE Trans. on Software Engineering* **17**(2) (1991) 141–152.
31. R. Selby and A. Porter, "Learning from examples: Generation and evaluation of decision trees for software resource analysis", *IEEE Trans. on Software Engineering* **14** (12) (1988) 1743–1757. Special Issue on Artificial Intelligence in Software Applications.
32. I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Tecniques with Java Implementations*, Morgan Kaufmann Publishers, 2000.
33. M. Woodward, M. Hennell, and D. Hedley, "A measure of control flow complexity in program text", *IEEE Transactions on Software Engineering* **5**(1) (1979) 45–50.

Appendix A. Metrics and Models for the Antenna Configuration System

List of the metrics collected for the empirical study:

[LOC]:	lines of code, excluding comment and blank lines.
[eLOC]:	effective lines of code, excluding comments, blank lines, and stand-alone braces or parenthesis.
[lLOC]:	logical lines of code, i.e., lines of code as identified by semi-colon.
[Comments]:	comment lines.
[Lines]:	all-inclusive count of lines of code.
[FP]:	number of formal parameters of functions.
[FR]:	number of return points of functions.
[IC]:	interface complexity, i.e., $FP + FR$.
[V(g)]:	cyclomatic complexity.
[CO]:	number of comparison operators.
[LFC]:	linear flow complexity, i.e., $CO + V(g)$.
[OC]:	operational complexity, i.e., weighted sum of operations for all occurring expressions.
[V'(g)]:	enhanced cyclomatic complexity, i.e., $V(g)/OC$.
[eV(g)]:	essential complexity.
[NEST]:	maximum number of nesting levels.

[Halstead's software science]:

[N1]:	total number of operators;
[N2]:	total number of operands;
[n1]:	number of unique operators;
[n2]:	number of unique operands;
[N]:	program length, i.e., $N1 + N2$;
[n]:	program vocabulary, i.e., $n1 + n2$;
[V]:	program volume, i.e., $N * \log 2n$;
[D]:	difficulty, i.e., $(n1/2) * (N2/n2)$;
[E]:	effort, i.e., $D * V$.

[CALLS]:	number of function calls.
[BRANCH]:	number of branching nodes.
[OAC]:	operation argument complexity, i.e., a weighted sum of the arguments in each operation.
[NION]:	number of input/output nodes.
[ANION]:	adjusted number of input/output nodes.
[CONTROL]:	number of control statements.
[EXEC]:	number of executable statements.
[NSTAT]:	number of statements, i.e., $CONTROL + EXEC$.
[CDENS]:	control density, i.e., $CONTROL/NSTAT$.

Relevant fault-proneness models selected in the empirical study:

Selection strategy	R^2	Metrics	C_i	p_i
Best R^2 (0.5272)	0.5272	eLOC	−0.82	<0.0001
		Comm	−0.44	0.0007
		Lines	0.48	<0.0001
		FP	0.44	0.0164
		LFC	−0.18	0.0141
		EXEC	0.33	0.0104
		Intercept	−4.63	<0.0001
Best overall completeness (89%)	0.4795	eLOC	−0.36	0.0001
		Comm	−0.37	0.0007
		Lines	0.34	<0.0001
		IC	0.38	0.0052
		NION	−0.96	0.0001
		Intercept	−3.23	0.0010
Best faulty module correctness (80%)	0.3685	eLOC	−0.09	0.0125
		Lines	0.09	0.0006
		FP	0.29	0.0167
		NION	−0.89	<0.0001
		Intercept	−1.22	0.0498
Best faulty module completeness (63%)	0.4884	eLOC	−0.37	0.0001
		Comm	−0.39	0.0004
		Lines	0.36	<0.0001
		FR	−1.05	0.0001
		IC	0.43	<0.0030
		Intercept	−4.40	<0.0001