JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 4
Special issue: TOOLS USA 2003

Improving Rule Set Based Software Quality Prediction: A Genetic Algorithm-based Approach

Salah Bouktif, Dept. of Computer Science and Op. Res., University of Montreal, Canada
Danielle Azar, **Doina Precup**, School of Computer Science, McGill University, Montreal, Canada
Houari Sahraoui and Balázs Kégl, Dept. of Computer Science and Op. Res., University of Montreal, Canada

Abstract
The object-oriented (OO) paradigm has now reached maturity. OO software products are becoming more complex which makes their evolution effort and time consuming. In this respect, it has become important to develop tools that allow assessing the stability of OO software (i.e., the ease with which a software item can evolve while preserving its design). In general, predicting the quality of OO software is a complex task. Although many predictive models are proposed in the literature, we remain far from having reliable tools that can be applied to real industrial systems. The main obstacle for building reliable predictive tools for real industrial systems is the lackof representative samples. Unlike other domains where such samples can be drawn from available large repositories of data, in OO software the lack of such repositories makes it hard to generalize, to validate and to reuse existing models. Since universal models do not exist, selecting an appropriate quality model is a difficult, non-trivial decision for a company. In this paper, we propose two general approaches to solve this problem. They consist of combining/adapting a set of existing models. The process is driven by the context of the target company. These approaches are applied to OO software stability prediction.

1 INTRODUCTION

Software quality assessment has become an increasingly important field. This fact is due to the determining role of software in today's world. The complexity caused by the emergence of new paradigms of object oriented (OO) design and programming makes the task, paradoxically, more important and more difficult. Indeed, OO software products are becoming more complex which makes their evolution effort and time consuming. In this respect, it has become important to develop tools that allow assessing the quality and particularly the stability of OO software (i.e., the ease with which a software item can evolve

Cite this article13 as follows: S. Bouktif, et al.: "Improving Rule Set Based Software Quality Prediction: A Genetic Algorithm-based Approach", in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pages 227–241, http://www.jot.fm/issues/issue_2004_04/

while preserving its design). Although many software quality models are proposed in the literature (see, for example, the study presented in [2]), OO software quality assessment remains unaccessible. The lack of reliable data for conducting empirical studies is one of the important reasons that explain this situation. In most of the domains where predictive models are built (such as sociology, medicine, finance, and speech recognition), researchers are free to use large data repositories from which representative samples can be drawn. In the area of OO software, however, such repositories are rare. The lack of data makes it hard to generalize, to validate and to reuse existing models. Since universal models do not exist, selecting an appropriate quality model is a difficult, non-trivial decision for a company.

An ideal predictive model can be seen as the mixture of two types of knowledge: common knowledge of the domain and context specific knowledge for a company. In the existing models, one of the two types is often missing. On the one hand, theoretical models [2] are designed to cover the common domain knowledge. Their application requires some adaptation/calibration to the particular context of a company. On the other hand, empirical models [4] contain the knowledge that was abstracted from a context-specific data set.

In this paper, we propose two genetic algorithm-based approaches to solve this problem. They consist in combining or adapting existing models (which we call *experts*). The process is driven by the context of the target company. These approaches are implemented for decision tree models and applied to OO software stability prediction.

The remainder of this paper is organized as follows: Section 2 describes the problem. In Section 3, we give an overview of the principles of genetic algorithms. The combination and the adaptation approaches are described in detail in Sections 4 and 5. Experimentation and discussion are given in Section 6.

2 PROBLEM STATEMENT

Most of the software quality factor (e.g., maintainability, reusability, reliability, stability), can not be measured until the software system has been used for a certain time. This fact motivated a wide range of studies aiming at an early prediction of these factors from some measurable software characteristics such as coupling, cohesion, and size [7]. Within this trend, a large number of OO metrics have been proposed in the literature (see [2],[1],[5] and [9]). In this paper, we are specifically interested in the prediction of OO software stability. During its operation time, a software undergoes various changes triggered by error detection, evolution in the requirements or environment changes. As a result, the behavior of the software gradually deteriorates as modifications increase. This quality slump may go as far as the entire software becoming unpredictable [11]. Consequently, there is a consensus that the software that is intended to last must remain stable in spite of requirement evolution. There are two ways of achieving this goal. Upstream, we can integrate the stability rules in the design phase as it is proposed in [10]. Downstream, a stability prediction model can be used to decide, after a certain number of versions, whether a

major refactoring is necessary to reduce the implementation cost of future requirements. Our current work is founded on this second belief. Hence, we study the prediction of the stability at the class level in OO software sytems. The key assumption in our study is that a class is stable whenever its interface remains valid between versions. Let c be a class and $I(c_i)$ be the interface of c in version i (public and protected, local and inherited methods). The level of stability of c can be measured by comparing $I(c_i)$ to $I(c_{i+1})$ (following version). It represents the percentage of $I(c_i)$ that is included in $I(c_{i+1})$[1]. Formally

$$ENS(c_i \longrightarrow c_{i+1}) = \frac{\#(I(c_{i+1}) \cap I(c_i))}{\#I(c_i)}.$$

Our hypothesis is that the stability of a class interface depends on the design (structure) of the class and the stress induced by the implementation of new requirements between the two versions. The predictive model will take the form of a function f that takes as input a set of structural metrics $(m_1(c_i), m_2(c_i), ..., m_n(c_i))$ and an estimation of the stress $St(c_i \longrightarrow c_{i+1})$ and produces as output a binary estimation of the stability $ENS(c) = ENS(c_i \longrightarrow c_{i+1})$ (1 for stable or -1 for unstable). Formally

$$ENS(c_i \longrightarrow c_{i+1}) = f(m_1(c_i), m_2(c_i), ..., m_n(c_i), St(c_i \longrightarrow c_{i+1})).$$

In this work, the stress $St(c_i \longrightarrow c_{i+1})$ represents the estimated percentage of added methods in c between the two versions. Formally

$$St(c_i \longrightarrow c_{i+1}) = \frac{\#(I(c_{i+1}) - I(c_i))}{\#I(c_{i+1})}.$$

The following is an example of a decision tree model (classifier) that predicts the stability of a class c using as input the two structural metrics $Coh(c)$ (class cohesion metric) and $NPPM(c)$ (number of public and protected methods), and the estimation of the stress $(St(c) = St(c_i \longrightarrow c_{i+1}))$.

Rule 1: $St(c) \leq 4\% \Rightarrow ENS(c) = 1$
Rule 2: $NPPM(c) \leq 19$ and $St(c) \leq 30\% \Rightarrow ENS(c) = 1$
Rule 3: $NPPM(c) > 3$ and $St(c) > 30\% \Rightarrow ENS(c) = -1$
Rule 4: $Coh(c) \leq 35\%$ and $NPPM(c) > 19$ and $St(c) > 4\% \Rightarrow ENS(c) = -1$
Default class: $ENS(c) = 1$.

This kind of classifiers are built using a data set $D_n = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ of n *examples* or *datapoints* where $\mathbf{x}_i \in \mathbb{R}^d$ is an *observation vector* of d attributes, and $y_i \in \mathcal{C} = \{c_1, ..., c_k\}$ a finite set of labels in the particular domain of our application, namely class stability. In this particular domain, an example $\mathbf{x}_i$ represents a class. The attributes of $\mathbf{x}_i$ are structural metrics and stress estimation. The classification label y_i of $\mathbf{x}_i$ represents the software quality factor to be predicted by the classifier - stability of a class, in our case. Hence y_i is a binary variable that takes the value 1 when the class is stable and -1 otherwise. The distribution of the data in the data set is unknown. The classifier

[1] We consider a deprecated method as if it is removed.

has a prediction accuracy rate that can be measured using the correctness of the classifier (the percentage of cases correctly classified) or its J-index (the average correctness per classification label). Our objective is to maximize the accuracy. For this, we are proposing two genetic algorithm-based approaches - one that combines different experts $f_1, f_2, ..., f_N$ and another one that adapts a single expert f_i to a set of data.

3 THE PRINCIPLE OF GENETIC ALGORITHMS

Genetic Algorithms (GA) were introduced in the late 1960's by John Holland [8]. They are based on the Darwinian theory of evolution whereby species compete to survive and the fittest get a higher chance to remain until the end and produce progeny. Typically, a GA starts with a population of individuals (chromosomes). Each individual is formed of genes and attributed a fitness value that measures how well it competes with the other chromosomes in the population. Roughly speaking, the process of evolution consists of passing from one population of chromosomes to the next by applying some genetic operators three of which are *crossover, mutation* and *selection*. During crossover, two chromosomes are selected, they exchange some of their genes giving birth to two other chromosomes. Mutation consists of changing randomly one or more genes in a chromosome. Both operators occur with a certain probability. Usually, the crossover operator occurs with a probability that is much higher than that at which mutation occurs. It is very possible that no crossover happens between a pair of chromosomes in which case the offsprings are exact copies of their parents. The offsprings resulting from crossover and the mutated chromosomes are copied to the next generation. The process is repeated a certain number of times or until a certain criteria is met. It is important to point out that a chromosome is selected to produce progeny with a certain probability that is proportional to its fitness.

4 A GENETIC ALGORITHM FOR COMBINING RULE SETS

This algorithm is designed specifically to combine rule sets into one final classifier. To apply a GA to this specific problem, elements of the generic algorithm must be instantiated and adapted to the problem. In particular, the solutions must be encoded into chromosomes and the two operators (crossover and mutation) and the fitness function must be defined. In the rest of this section, we present the implementation of each of these elements for the algorithm.

Model Encoding

A decision tree is a complete binary tree where each inner node represents a yes-or-no question, each edge is labeled by one of the answers, and terminal nodes contain one of the classification labels from the set $\mathcal{C}$. The decision making process starts at the root of

the tree. Given an input vector **x**, the questions in the internal nodes are answered, and the corresponding edges are followed. The label of **x** is determined when a leaf is reached.

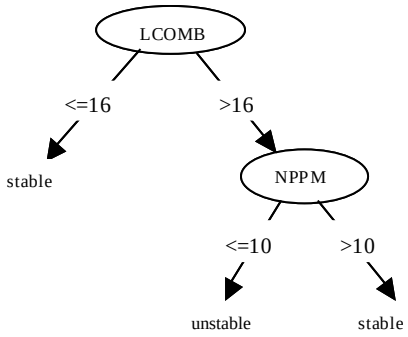


Figure 1: A decision tree for stability prediction.

If all the questions at the inner nodes are of the form "Is $x^{(j)} > \alpha$?" (as in the tree depicted in Figure 1), the decision regions of the tree can be represented as a set of isothetic boxes (boxes with sides parallel to the axes) in an n-dimentional space (n being the number of metrics used in the decision tree). Figure 2 shows this representation of the tree in Figure 1. To represent the decision trees as chromosomes in the GA, we enumerate these decision regions in a vector. Formally, each gene is a (box,label) pair where the box $b = \left\{\mathbf{x} \in \mathbb{R}^d : \ell_1 < x^{(1)} \leq u_1, \ldots, \ell_d < \mathbf{x}^{(d)} \leq u_d\right\}$ is represented by the vector $\big((\ell_1, u_1), \ldots, (\ell_d, u_d)\big)$, and a vector of these (box,label) pairs constitutes a chromosome representing the decision tree. To close the opened boxes at the extremities of the input domain, for each input variable $x^{(j)}$, we define lower and upper bounds L_j and U_j, respectively. For example, assuming that in the decision tree of Figure 1 we have $0 < NPPM(c) \leq 100$ and $0 < LCOMB(c) \leq 50$, the tree is represented by the nested vector

$$\left(\begin{array}{l} \big((0,10),(16,50);-1\big), \\ \big((10,100),(16,50);1\big), \\ \big((0,100),(0,16);1\big) \end{array} \right).$$

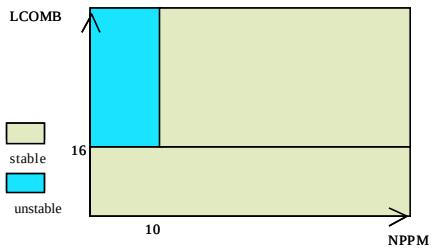


Figure 2: A two-dimensional example of decision tree output regions.

The Crossover Operator

A standard way to perform the crossover operation between the chromosomes is to cut each of the two parent chromosomes into two subsets of genes (boxes in our case). Two new chromosomes are created by interleaving the subsets. If we apply such an operation in our problem, it is possible that the resulting chromosomes can no longer represent well-defined decision functions. Two specific problems can occur. If two boxes overlap, we say that the model is *inconsistent*. In this case, the model represented by the chromosome is not even a function. The second problem is when the model is *incomplete*, that is, certain regions in the input domain are not covered by any box. Figure 3 illustrates these two situations.

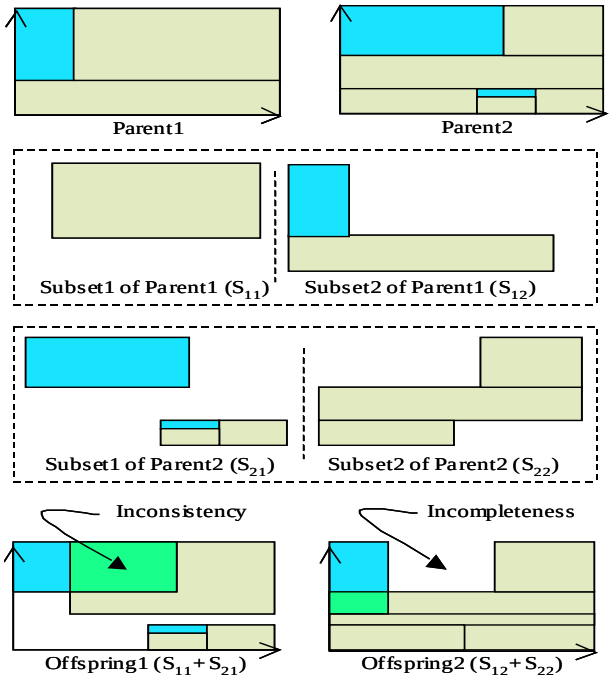


Figure 3: Problems when using standard crossover.

To preserve the consistency and the completeness of the offspring, we propose a new crossover operator inspired by the operator defined for grouping problems [6]. To obtain an offspring, we select a random subset of boxes from one parent and add it to the set of boxes of the second parent. The size of the random subset is v times the number of boxes of the parent where v is a parameter of the algorithm. By keeping all the boxes of one of the parents, completeness of the offspring is automatically ensured. To guarantee consistency, we make the added boxes predominant (the added boxes are “laid over” the original boxes). Figure 4 illustrates the new crossover operator.

A *level of predominance* is added as an extra element to the genes. Therefore, each gene is now a three-tuple (box, label, level). The boxes of the initial population P_0 have level 1. Each time a predominant box is added to a chromosome, its level is set to 1 plus the maximum level in the hosting chromosome. To find the label of an input vector x (a

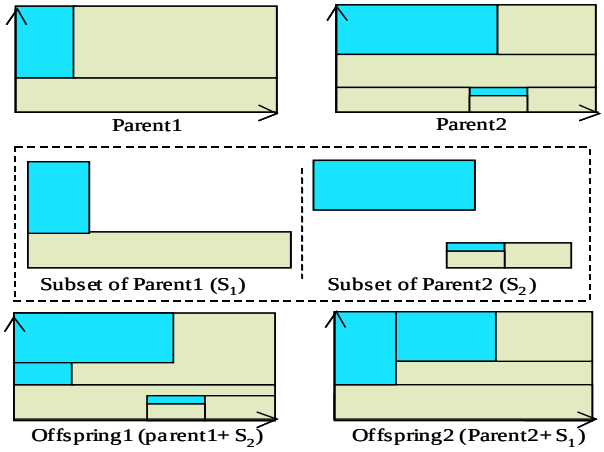


Figure 4: Crossover that preserves consistency and completeness.

software element), first we find all the boxes that contain x, and assign to x the label of the box that have the highest level of predominance. This scheme is similar, in spirit, to rule systems where rules have priorities that are used to resolve conflicts among them.

The Mutation Operator

Mutation is a random change in the genes that happens with a small probability. In our problem, the mutation operator randomly changes the label of a box. In software quality prediction, the output space $\mathcal{C}$ is usually an ordered set $c_1,\ldots,c_k$ of labels. With probability p_m, a label c_i is changed to c_{i+1} or c_{i-1} if $1 < i < k$, to c_2 if $i = 1$, and to c_{k-1} if $i = k$.

In general, other types of mutations are possible. For example, we could change the size of a box, or we could set the label of a box to the label of an adjacent box. The main reason of our choice of the mutation operator is its simplicity.

The Fitness Function

To measure the fitness of a decision function f represented by a chromosome, one could use the *correctness function*

$$C(f) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k}\sum_{j=1}^{k} n_{ij}},$$

where n_{ij} is the number of training vectors with real label c_i classified as c_j (Table 1). It is clear that $C(f) = 1 - L(f)$ where $L(f)$ is the training error.

Software quality prediction data is often *unbalanced*, that is, software components tend to have one label with a much higher probability than other labels. For example, in our experiments we had many more stable than unstable classes. On an unbalanced data set, low training error can be achieved by the constant classifier function f_{const} that assigns

		predicted label			
		c_1	c_2	...	c_k
real label	c_1	n_{11}	n_{12}	...	n_{1k}
	c_2	n_{21}	n_{22}	...	n_{2k}
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	c_k	n_{k1}	n_{k2}	...	n_{kk}

Table 1: The confusion matrix of a decision function f. n_{ij} is the number of training vectors with real label c_i classified as c_j.

the majority label to every input vector. By using the training error for measuring the fitness, we found that the GA tended to "neglect" unstable classes. To give more weight to data points with minority labels, we decided to use Youden's *J-index* [13] defined as

$$J(f) = \frac{1}{k}\sum_{i=1}^{k}\frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}.$$

Intuitively, $J(f)$ is the average correctness per label. If we have the same number of points for each label, then $J(f) = C(f)$. However, if the data set is unbalanced, $J(f)$ gives more relative weight to data points with rare labels. In statistical terms, $J(f)$ measures the correctness assuming that the a priori probability of each label is the same. Both a constant classifier f_{const} and a guessing classifier f_{guess} (that assigns random, uniformly distributed labels to input vectors) would have a J-index close to 0.5, while a perfect classifier would have $J(f) = 1$. On the other hand, for an unbalanced training set, $C(f_{\text{guess}}) \simeq 0.5$ but $C(f_{\text{const}})$ can be close to 1.

5 A GENETIC ALGORITHM FOR ADAPTING A RULE SET

This is an adaptive approach whereby one rule set is evolved into another by adapting it to a training set (context). This approach will be compared to the combining one. In this section, we will implement the different elements of GA, as it is done in Section 4, according to the aspects of the problem of adapting a rule set.

Chromosome Encoding and Fitness Function

We start with one rule set which constitutes the initial population of chromosomes, each rule of the rule set being a chromosome and each condition in the rule as well as the classification label being a gene. For example, the following rule set constitutes a population of chromosomes where the first one is formed of four genes, each of the first three encoding a condition and the last gene encoding the classification label.

Rule1: $NOC(c) < 6$ and $CUB(c) > 2$ and $NMI(c) > 1 \Rightarrow ENS(c) = 1$

Rule2: $NOC(c) < 8$ and $COH(c) > 1 \Rightarrow ENS(c) = -1$
Default class: $ENS(c) = 1$.

The default classification in the rule set is replaced with one or more explicit rules. Each chromosome i is attributed a fitness value $fr(i) = C(i) * t(i)$ where $C(i)$ is the correctness of the rule that chromosome i encodes (it is equal to the number of cases that the rule correctly classifies divided by the total number of cases that the rule classifies) and $t(i)$ is the fraction of cases that the rule classifies in the training set (number of cases that the rule classifies divided by the total size of the training set). The weight $t(i)$ allows to give rules that cover a large set of training cases, a higher chance of being selected. Correctness is defined in more detail in Section 4.

The Genetic Operators

The process of evolution starts by selecting two chromosomes according to the *roulette-wheel* technique [8], whereby one can imagine a roulette wheel where all chromosomes are placed. Each chromosome is assigned a portion of the wheel that is proportional to its fitness. A marble is thrown and the chromosome where the marble halts is selected. A random cut point is generated for each chromosome in the selected pair and two offsprings are generated. The first one receives the genes from its first parent up to the cut point along with the genes of its second parent starting at the cut point. The second offspring combines the first part of the second parent along with the second part of the first parent. For example, applying crossover to the chromosomes shown above (Rule1 and Rule2) and defining the first cut point to be at the second gene in the first parent and the second cut point to be at the first gene of the second parent, the two offspring resulting from this operation are:

Rule3: $NOC(c) < 6$ and $NOC(c) < 8$ and $COH(c) > 1 \Rightarrow ENS(c) = -1$ (offspring 1)
Rule4: $CUB(c) > 2$ and $NMI(c) > 1 \Rightarrow ENS(c) = 1$ (offspring 2).

By allowing chromosomes within a pair to be cut at different places, we allow for a wider variety with respect to the length of the chromosomes (and hence, the rules that they encode). Before being thrown in the next generation, the chromosomes are mutated with a certain probability. Mutation of a gene consists of changing the value to which the attribute encoded in the gene is compared to a value chosen randomly from a predefined set of values for the attribute (or class label, in case the last gene is mutated). For example, the second offspring shown above (Rule 4) can be mutated by changing the value in the second gene to 2.5 and the chromosome becomes:

Rule4: $CUB(c) > 2$ and $NMI(c) > 2.5 \Rightarrow ENS(c) = 1$ (offspring 2 mutated).

The new chromosomes are then scanned and trimmed to get rid of any redundancy in the conditions that form the rules that they encode. For example, offspring 1 is trimmed

into the rule: $NOC(c) < 6$ and $COH(c) > 1 \Rightarrow ENS(c) = -1$.

Inconsistent rules (for example, $NOC(c) > 4$ and $NOC(c) < 2 \Rightarrow ENS(c) = 1$) are attributed a fitness value of 0 and will eventually die. Our algorithm maintains a fixed size of population throughout the process of evolution. In order to ensure this, *elitism* is performed when the population size is odd. Elitism consists of copying one or more of the best chromosomes from one generation to the next. This ensures that the best individuals are not lost during the process of evolution. After crossover and mutation are performed, the GA checks the flag for elitism. If this is ON, the best chromosome in the current generation is selected and thrown in the next generation, if the flag is OFF, a chromosome is chosen randomly and thrown in the next generation. If the size of population is even, no elitism takes place.

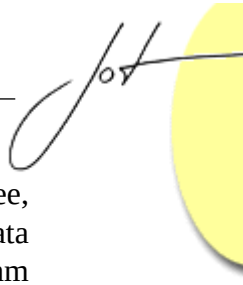
Combining the Rules into One Rule Set

Before passing from one generation of chromosomes to the next, the GA evaluates how well the rules perform when they are combined together into one rule set. For this, a rule set is formed out of these rules and attributed a default classification label. This is the majority classification label in the training set. The rule set is then evaluated on the training set by computing its J-index. If the rule set is at least as good as the one formed in the previous population, it is kept, otherwise, it is discarded and the previous rule set replaces it. This way, we ensure that the performance of the rule set on the training data does not deteriorate. J-index is defined in more detail in Section 4. The GA runs for a pre-determined number of iterations. At the end, the resulting rule set is compared to the initial rule set by computing the J-index of both on a set of unseen data.

6 EXPERIMENTATION

Experimental Settings

In order to empirically validate our approaches, we constructed a ”semi-real” environment in which the ”in-house” data set is a real software system with several versions, but the experts are ”simulated”: they are decision tree classifiers trained on independent software system data. To imitate the heterogeneity of real life experts, each expert was trained on a different sub-set of metrics and on a different software system. Although we are aware of the limitations of this model, we found that it simulated reasonably well a realistic situation and yielded some interesting results. To build the experts (that simulate the existing models), we use the stress (c.f. Section 2) plus 18 structural metrics that belong to one of the four categories of coupling, cohesion, inheritance, and complexity (see Table 2). The detailed definitions of these metrics and the predictive models where they were used can be found in [5],[2],[14],[4]and [3]. The metrics were extracted from 11 OO systems (see Table 3). These systems were used to ”create” 40 experts in the following



way: First, we formed 15 subsets of the 19 software metrics by combining two, three, or four of the metric categories in all the possible ways, and created 15 x 11 = 165 data sets. Then, we trained a decision tree classifier on each data set using the C4.5 algorithm [12]. We retained 40 decision trees by eliminating constant classifiers and classifiers with training error more than 10%. The company specific context is represented by standard JAVA API[2]. The four first major versions were used. We created a data set D_n of 2920 data vectors using the classes in the four versions (see Table 4).

Name	Description
Cohesion metrics	
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
Inheritance metrics	
NOC	number of children
NOP	number of parents
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted meth. per cl.
NPPM	number of public and protected meth. in a cl.
NPA	number of public attributes

Table 2: The 18 software metrics used as attributes in the experiments.

For the algorithmic settings, as a meta-heuristic algorithm, the GA has several parameters chosen on an experimental basis. For the combining GA (Section 4), the elitist strategy was used: In each iteration, the entire population was replaced, except for a small number N_e of the fittest chromosomes. The number of generations T was set to 100. The maximum number of chromosomes in a generation was set to 160 in order to have a reasonable execution time. The values of N_e, p_c (crossover probability), p_m (mutation probability), and v (proportion of the random subset of boxes used in the crossover operation) change during the evolution of the number of created generations t as shown in

[2] Available at http://java.sun.com/api/index.html.

System	Number of (major) versions	Number of classes
Bean browser	6(4)	388–392
Ejbvoyager	8(3)	71–78
Free	9(6)	46–93
Javamapper	2(2)	18–19
Jchempaint	2(2)	84
Jedit	2(2)	464–468
Jetty	6(3)	229–285
Jigsaw	4(3)	846–958
Jlex	4(2)	20–23
Lmjs	2(2)	106
Voji	4(4)	16–39

Table 3: The software systems used to train the experts.

JDK version	Number of classes
jdk1.0.2	187
jdk1.1.6	583
jdk1.2.004	2337
jdk1.3.0	2737

Table 4: The software systems (context) used to combine/adapt the experts.

(Table 5). In the adaptating GA (Section 5), elitism consists in choosing the best chromosome and copying it to the next generation.The best value for T was found to be 100, 0.85 for p_c and 0.2 for p_m.

t	0–10	11–30	31–99
N_e	3	5	10
p_c	0.65	0.65	0.60
p_m	0.02	0.03	0.05
v	0.3	0.1	0.05

Table 5: Combining GA parameters.

Results

To accurately estimate the J-index of the trained classifiers, we used 10-fold cross validation. In this technique, the data set is randomly split into 10 subsets of equal size (292 points in our case). A decision function is trained on the union of 9 subsets, and tested on the remaining subset. The process is repeated for all 10 possible combinations. Mean and standard deviation values are computed for J-index on both the training and the test samples. Table 6 shows our results. We used the simple selection of the best expert according

to the context (f_{best}) as a benchmark for evaluating our approaches. The final experts resulting from the combination and the adaptation are called, respectively, $f_{\text{combining}}$ and f_{adapting}.

		Training	Test
J-index $J(f)$	f_{best}	66.50(0.50)	65.98(4.25)
	f_{adapting}	67.21(0.90)	67.34(3.85)
	$f_{\text{combining}}$	78.24(2.43)	76.86(5.06)

Table 6: Experimental results. The mean (standard deviation) percentage values of the J-index.

The test results show that our approach of combining experts can yield significantly better results than using individual models. The adaptation approach did not perform as well as the combination, although it gave a slight improvement over the initial model generated by C4.5. The main reason of this difference is the fact that each expert uses only a subset of the software metrics and does not cover the complete common domain knowledge as the combination approach does. We strongly believe that if we use more numerous and real experts on cleaner and less ambiguous data, the improvement will be more significant.

7 CONCLUSION

Assessing the quality of OO software is a complex task. As evidence, we can simply look at the very few models that are concretely used in industry compared to the considerable number of models proposed in the literature. The lack of reliable data for conducting serious empirical studies is, in our opinion, the main obstacle to obtaining satisfactory results in this domain. Moreover, even when a predictive model is theoretically sound, it is rare that it is adapted to the context of a specific company. To circumvent these two problems, in this paper, we propose, two evolutionary approaches for combining and adapting existing OO software quality predictive models to a particular context. In the case of the combination approach, the resulting model can be interpreted as a ”meta-expert” that selects the best expert for each given case. This notion corresponds well to the ”real world” in which individual predictive models, coming from heterogeneous sources, are not universal and depend largely on the underlying data. In the case of adaptation approach, the resulting model can be seen as a calibrated model. Using these two approaches, we conducted experiments on the prediction of Java class stability. The preliminary results show that the combination of models can perform significantly better than simply choosing an existent one. For the adaptation approach, we cannot draw a final conclusion because the improvement is not statistically significant. Issues of future research include the evaluation of the approaches on real experts proposed in the literature (e.g., default estimation models). For the adaptation approach, alternative local search algorithms (e.g., tabu search and simulated annealing) will be investigated.

REFERENCES

[1] J. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. *ACM Symp. Software Reusability*, 1995.

[2] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[3] L. Briand and J. Wüst. Empirical studies of quality models in object-oriented systems. In M. Zelkowitz, editor, *Advances in Computers*. Academic Press, 2002.

[4] L. Briand, J. Wüst, J. W. Daly, and V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51:245–273, 2000.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions of Software Engineering*, 20(6):476–493, 1994.

[6] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, 1998.

[7] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex sofware system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[8] J. H. Holland. *Adaptation in Natural Artificial Systems*. University of Michigan Press, 1975.

[9] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[10] R. Martin. Stability. c++ report. 9(2), 1997.

[11] D. Parnas. Software aging. In *16th International Conference on Software Engineering*, 1994.

[12] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[13] W. J. Youden. How to evaluate accuracy. *Materials Research and Standards, ASTM*, 1961.

[14] H. Zuse. *A Framework of Software Measurement*. Walter de Gruyter, 1998.

ABOUT THE AUTHORS

Salah Bouktif is a Ph.D. student at the department of computer science and operational research of the University of Montreal. His research interests relate to the software quality, metrics and software prediction models. Salah can be reached at bouktifs@iro.umontreal.ca.

Danielle Azar (dazar@cs.mcgill.ca) is a Ph.D. candidate in the School of Computer Science at McGill University in Montreal, Canada. Her main areas of interest include genetic algorithms and their application in software engineering, particularly in the optimization of software quality estimation models.

Houari Sahraoui received a Ph.D. in Computer Science from Pierre & Marie Curie University, Paris in 1995. He is currently an associate professor at the department of computer science and operational research, University of Montreal where he is leading the software engineering group. His research interests include object-oriented software quality and software reverse and re-engineering. He can be reached at sahraouh@iro.umontreal.ca.

Balázs Kégl received the Ph.D. degree in computer science (with honors) from Concordia University, Montreal, Canada, in 1999. His is currently with the Department of Computer Science and Operational Research at the University of Montreal as an assistant professor. His research interests include statistical pattern recognition, machine learning, and image processing. He can be reached at kegl@iro.umontreal.ca.

Doina Precup received her PhD in Computer Science from the University of Massachusetts, Amherst,in 2000. In July 2000 she joined the School of Computer Science at McGill University. Doina Precup's research interests lie mainly in the field of machine learning. She is especially interested in the learning problems that face a decision-maker interacting with a complex, uncertain environment. She can be reached at dprecup@cs.mcgill.ca.