CHAPTER 25

AN ADAPTIVE APPROACH TO OPTIMIZE SOFTWARE COMPONENT QUALITY PREDICTIVE MODELS: CASE OF STABILITY

DANIELLE AZAR[1] AND DOINA PRECUP[2]

[1]*Lebanese American University, Computer Science and Math Division, P.O.Box 36, Byblos, Lebanon*
[2]*McGill University, School of Computer Science, 3480 University St., Montreal, Quebec, Canada, H3A 2A7*

Abstract: Assessing software quality is fundamental in the software developing field. Most software characteristics (such as maintainability, reusability, stability, etc.) cannot be measured before the software product is used for a certain period of time. However, these characteristics can be predicted or estimated based on other measurable software attributes (such as cohesion, coupling, and size). For this purpose, software quality estimation models have been extensively used. In particular, rule-based software quality estimation models have been popular because of their white-box nature. Their usefulness is two-fold: 1) they give the prediction itself, 2)they give guidelines to attain it. However, the data used to build such models is normally scarce. As a result, the accuracy of these models deteriorates when they are used on new/unseen data. As a consequence, it is difficult to generalize, to cross-validate and to reuse existing models. In this work, we propose an approach to combine and adapt already-existing models to a new set of data. Our approach is evolutionary and relies on the genetic algorithm methodology. The "goodness" of the new model is measured by how well it classifies new/unseen data. Results show that our technique outperforms C4.5 and random guess when tested on stability of object-oriented software components

1. INTRODUCTION

Object-Oriented (OO) software products are becoming more and more complex. It is becoming crucial that the quality of the software be evaluated during the different stages of the development. The quality of a software is evaluated in terms of characteristics such as maintainability, reusability, stability, etc. Most of these characteristics cannot be measured before the software is used for a certain period of time. Software attributes (such as cohesion, coupling and size), on the other hand, can be directly measured and at the same time, can be used as indicators of

H. Labiod and M. Badra (eds.), New Technologies, Mobility and Security, 297–310.

these characteristics. For this purpose, several metrics that capture such attributes have been proposed in the literature [1–9]. Software quality estimation models build a relationship between the desired software quality and the measurable attributes. These models are either statistical models or logical models. The latter have been extensively used because of their interpretability. Practitioners want to know the reason of the assessment as well as the prediction of the model. Logical models can take the form of decision trees or rule sets. Rule sets are easier to read by human experts. However, logical models, in general, suffer from degradation of their prediction when they are applied to new/unseen data. This is largely due to the lack of a representative sample that can be drawn from available data in the domain of software quality. Unlike other fields where public repositories abound with data, software quality data is usually scarce. The reason behind the scarcity of the data is two-fold: 1. There are not many companies that systematically collect information related to software quality. 2. Such information is normally considered confidential. In the case where a company is willing to make it public, usually only the resulting model is published. The latter is company-specific. The lack of data behind it makes it difficult to generalize, to cross-validate or to re-use this model in other companies. In this paper, we present an approach to build a better model from already-existing models. This can be seen as combining the expertise of several expert models, built from common domain knowledge, and adapting the resulting models to context-specific data. The approach relies on the genetic algorithm (GA) paradigm. To validate our technique, we use the specific problem of predicting the stability of object-oriented (OO) software components (classes, in this context). During its evolution, a software component undergoes various modifications due to changes in requirements, error detections, changes in the environment, etc. It is important for the software component to remain "usable" in the new changed environment. Briefly, a class in an object-oriented system is said to be *stable* if its public interface remains valid between different versions of the system; otherwise, it is said to be *unstable*. The remainder of this paper is organized as follows. In Section 2, we introduce the notation used to describe the problem. In Section 3, we give a brief overview of genetic algorithms. In Section 4 we describe our evolutionary approach. In Section 5 we describe the experiments that we conducted and the results we obtained and we conclude in Section 6 with future work.

2. PROBLEM FORMULATION

In this section we state the problem in hands and we introduce the formalism that we use throughout the paper. The notation originates from the machine learning formalism.

A *data set* is a set $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ of *n instances* or *cases* where $x_i =< a_1, a_2, ..., a_d > \in \Re^d$ is an *attribute vector* of *d attributes*, and $y_i \in C$ is a *classification label*. In the particular problem that we are considering, a case represents a software component (a class in an OO software system). The attributes $(a_1, ..., a_d)$ are metrics (such as number of methods, number of children, etc.)

that are considered to be relevant to the software quality factor being predicted (stability). The label y_i represents the software quality factor. In this problem, $y_i \in \{0(stable), 1(unstable)\}$.

A *classifier* is a function $f : \Re^d \mapsto C$ that predicts the label y_i of any attribute vector x_i. In the framework of supervised learning, it is assumed that (vector, label) pairs are random variables (X, Y) drawn from a fixed but unknown probability distribution, and the objective is to find a classifier f with a low error (misclassification) rate. Since the data distribution is unknown, both the selection and the evaluation of f must be based on the data set D. For this purpose, D is partitioned into two parts, the *training set* D_{train} and the *testing set* D_{test}. Most learning algorithms take the training set as input and search the space of classifiers for one that minimizes the error on D_{train}. The output classifier is then evaluated on the testing sample D_{test}. Examples of learning algorithms that use this principle are the back propagation algorithm for feed forward neural nets [10] and C4.5 [11]. In our experiments, we use *10-fold cross validation* that allows us to use the whole data set for training and to evaluate the error probability more accurately. This is discussed in more detail in Section 5.

In our work, we focus on rule-based classifiers i.e classifiers that take the form of rule sets. A rule-based classifier is a disjunction of conjunctive rules and a default classification label. The following example illustrates a rule-based classifier that predicts the stability of a component (a class in an object-oriented software system) based on the metrics *CUBF* (number of classes used by a member function), *NOP* (number of parents) and *CUB* (number of used classes).

$Rule1:$ $CUBF > 7 \rightarrow 1$
$Rule2:$ $NOP > 2 \rightarrow 1$
$Rule3:$ $CUB \leq 2 \wedge NOP \leq 2 \rightarrow 0$
$Default\ class:$ 1

This model has three rules and a default classification label. The first rule estimates that if the number of classes used by a member function (CUBF) in the designated class is greater than 7 then the class is unstable (1). The second rule classifies a class with number of parents (NOP) greater than 2 as unstable. The third rule classifies a class with the number of used classes (CUB) less than or equal to 2 and number of parents (NOP) less than or equal to 2 as stable (0). The classification is sequential. The first rule (toward the top) whose left hand side is satisfied by a case fires. If no such rule exists, the default classification label is used to classify the case (default class 1).

The model has an accuracy that can be measured using the correctness of the classifier (percentage of cases correctly classified) or its J_index(average correctness per class label). The correctness of the rule set R computed on a data set D is equal to the percentage of cases in D that are correctly classified by $R(\ 1)$. In the equation, n_ij is the number of cases in D that are classified by R as having class

label j while they actually have class label i. The J_index is the average correctness per class label(2).

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

$$J(R) = \frac{1}{k} \sum_{i=1}^{k} \frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}} \tag{2}$$

Intuitively, if we have the same number of instances of each label, then $J(R) = C(R)$. However, if the data set is unbalanced, $J(R)$ gives more relative weight to cases with minority labels. Both a constant classifier and a guessing classifier (that assigns random uniformly distributed labels to input vectors) would have a J_index close to 0.5, while a perfect classifier would have a J_index equal to 1. On the other hand, in the case of an unbalanced data set, $C(R) \approx 0.5$ but $C(const)$ can be close to 1. Our goal is to maximize the accuracy on the unseen data. Next, we present our genetic algorithm-based approach that combines different classifiers and adapt the resulting ones to a new data set.

3. AN OVERVIEW OF GENETIC ALGORITHMS

Genetic algorithms were introduced in the early 70's by John Holland [12]. They are based on the Darwinian theory of evolution [13] whereby individuals compete and the fittest have a higher change to survive and produce progeny. Typically, a genetic algorithm (GA) starts with a population of individuals or chromosomes (solutions to a problem). Each individual is attributed a fitness value that indicates the quality of the underlying solution. The GA iterates to create new populations (of fitter chromosomes, ideally) through the application of genetic operators on the chromosomes in the current population. The basic operators are crossover, mutation and elitism. Roughly speaking, a chromosome is formed of genes. During crossover, two chromosomes exchange some of their genes and produce two other chromosomes. Crossover happens with a certain probability (usually 50% and above). In case this operation is not performed, the offspring are exact copies of their parents. One or more genes in the offspring are changed randomly by the application of the mutation operator. Mutation also happens with a certain probability (usually 0.1% and below). The offspring are then copied into the new population. The best chromosomes found in a population are preserved by the elitism operator (by copying them as is to the next population). Usually, population at time $t+1$ has size equal to size of population at time t. The whole process of creating new populations is repeated until a certain stopping criteria is met (after a pre-determined number of iterations or a desired fitness value is achieved). It is important to point out that selection of the chromosomes happens with a probability proportional to their fitness.

4. A GENETIC ALGORITHM-BASED APPROACH FOR COMBINING RULE SETS

Our approach relies on the idea of combining different classifiers built from some source of knowledge (common domain, for example) and adapting the resulting model(s) to a new/unseen data set (company-specific data). Combining classifiers is a difficult problem due to the size of the search space. Exhaustive and local search methods are inefficient since the problem involves a large set of classifiers that use different metrics. Genetic algorithms offer an interesting approach to this problem. The basic idea is to form a population of chromosomes (a pool of classifiers). New populations are created by combining and modifying classifiers to obtain new (ideally better) ones. In the rest of this section, we describe how we instantiate the elements of the GA (chromosome encoding, genetic operators and fitness function).

4.1. Model Coding and Fitness Function

The coding is straightforward. In our problem, a chromosome represents a rule-based classifier (rule set). Each condition and class label is encoded as a gene in the chromosome. They are listed in the order of their appearance in the rule set (Fig. 1). The objective function to optimize is the correctness of the rule set on the training data. So, the fitness of a chromosome is equal to the correctness of the classifier that the chromosome encodes.

4.2. Elitism, Crossover, and Mutation

The GA starts by copying a percentage of the chromosomes to the next population (elitism)[1] then the population is completed by crossover and mutation. Two chromosomes are selected[2] and a cut point is generated randomly in each of them. The offspring

Rule set
Rule :
NMO>1 AND NMI>22 AND SIX>1 − >class 0
Rule2:
NOD>8 AND NMI>22 − > class 1
Default class: 0
Chromosome
NMO>1 NMI>22 SIX>1 C0 NOD>8 NMI>22 C1 C0
gene1 gene4

Figure 1. A rule set and its chromosome representation: Each rule and condition in the rule set is represented in one gene in the chromosome

[1] This percentage is a parameter input to the algorithm.
[2] Fitter chromosomes get a higher probability of being selected.

Parent1
NMO>1 NMI>22 SIX >1 C0 NOD >8 NMI >22 C1 C0
Parent2
DIT>2 NMI>22 C1 NOD>8 C0 C1
Offspring 1
NMO>1 NMI>22 C1 NOD>8 C0 C1
Offspring 2
DIT>2 NMI>22 SIX>1 C0 NOD>8 NMI>22 C1 C0

Figure 2. Crossover where cutpoints (arrows) fall within a rule in chromosomes

are formed by exchanging the tails (sections after the cutpoints) of the parents. A cutpoint can fall within a rule or on a rule boundary (between two rules or between a rule and the default classification label) in a chromosome. In order to insure the validity of the resulting chromosomes, we apply the following "tuning" to our crossover operator: If cut point i_1 falls within a rule in parent p_1, cut point i_2 has to fall within a rule in parent p_2. If i_1 falls on a rule boundary in p_1, i_2 has to fall on a rule boundary in p_2. Moreover, the operator is implemented in such a way that cut points are not allowed to fall on chromosome boundaries (Fig. 2 and 3). The offspring chromosomes are mutated before being copied into the next population. Each one is scanned and each gene is changed with a certain probability. If the gene represents a condition ($NOC \leq 5$, for example), mutation consists of changing the value to which the attribute is compared to a value chosen randomly from the set of cutpoints of the attribute[3] ($NOC \leq 2$, for example). We chose cutpoints because these are more likely to change the set of cases that the rule classifies [14]. If the gene encodes a classification label, mutation consists of changing this value to one taken randomly from the domain of the classification labels (changing 0 to 1 for example).

Parent1
NMO>1 NMI>22 SIX >1 C0 NOD >8 NMI >22 C1 C0
Parent2
DIT>2 NMI>22 C1 NOD>8 C0 C1
Offspring 1
NMO>1 NMI>22 SIX>1 C0 NOD>8 C0 C1
Offspring 2
DIT>2 NMI>22 C1 NOD>8 NMI>22 C1 C0

Figure 3. Crossover where cutpoints (arrows) fall on a rule boundary in chromosomes

[3] A cutpoint of an attribute is the median of the two values of the attribute where the classification label changes [11].

4.3. Trimming

The operators described above can result in rule sets that contain *redundancy* and *inconsistency*. Both can occur within a rule or among rules. Redundancy inside a rule occurs when the rule has two conditions c_i and c_j where c_i implies c_j (example, c_i is $NOC \leq 4$ and c_j is $NOC \leq 6$). The offspring are scanned and $\forall i, j$, if c_i implies c_j, c_j is dropped. Redundancy can also occur among rules when two or more rules in the rule set have the same conditions (the order is not important). This is eliminated by keeping one of the rules only. Inconsistency, on the other hand, is allowed within a rule as long as evolution is ongoing. A rule is inconsistent when it has two conditions c_i and c_j that cannot be true at the same time (example, c_i is $NOC \leq 3$ and c_j is $NOC > 5$). Inconsistency is allowed because several experiments have shown that some conditions, although "fatal" in some combination, can have good impact when combined with other conditions. Inconsistency among rules is retained. Contradicting rules are allowed to remain in the rule set as one of them only will contribute to the classification of the rule set.

5. EXPERIMENTATION

The software quality that we experimented with is the stability of OO components (classes in an OO software system). We say that a component is *stable* when its public interface remains unchanged between two consecutive versions of the class. Otherwise, we say that the class is *unstable*. Stability of a class is affected by the structure of the class and the stress induced by the implementation of new requirements (these requirements are typically added methods) between two versions [15]. For this reason, we chose structural metrics that belong to one of the four categories: cohesion, coupling, inheritance, and size complexity as well as the stress metric. We validate our approach on the data sets described in [15] and [16].

5.1. Data Collection

STAB1[16]- Nineteen structural metrics (Table 1) were extracted from the eleven software systems[4] shown in Table 2 using the ACCESS tool and the Discover environment©[5]. Detailed description of these metrics can be found in [2], [17], [18], [3], and [19]. Fifteen subsets (of size 2, 3, and 4) were created by combining these metrics in all possible ways. The combination was based on the relationship desired among the different quality characteristics[6]. The 15 subsets of metrics were used with the 11 software systems to create $15X11 = 165$ data sets. C4.5 was

[4] JDK versions are available at http://java.sun.com/api/index.html.
[5] Available at http://www.mks.com/products/discover/developer.shtml.
[6] For example, the relationship between cohesion and coupling on one hand and stability on the other.

Table 1. STAB1-Software quality metrics used as attributes in the classifiers

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
The stress metric	
STRESS	stress applied to the class

used to create 165 decision tree classifiers with these data sets. Constant classifiers (classifiers that have a single classification label) and classifiers with a training error more than 10% were eliminated and the remaining 40 were retained. These decision trees were then converted into rule sets by C4.5. Our GA uses these rule sets as experts built from common domain knowledge and combines and adapts them to the four software systems shown in Table 3. These form a data set which consists of 2920 instances and simulate the in-house data.

STAB2[15]- The previous data set is highly imbalanced. As a matter of fact, 2481 cases are stable and 439 unstable. In order to test our algorithm on a balanced data set, we used STAB2 which also involves stability. Table 6 shows the 22 software metrics used. Nine of the 11 software systems shown in Table 2 were used to build experts with C4.5 (Table 4). Fifteen subsets of metrics were created by combining 2, 3, or 4 groups in all possible ways. These subsets were used with the 9 chosen software systems to create 135 data sets. C4.5 was used to construct a decision tree from each data set. Constant classifiers and classifiers with an error rate higher than 10% were eliminated and 23 retained. The decision trees were then converted to rule sets by C4.5. Our GA uses the systems shown in Table 5 to train and test the GA (in-house data simulation).

Table 2. STAB1-Software systems used to build classifiers with C4.5

Software System	Number of versions (major)	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. STAB1–Software systems used to train and test the GA

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

Table 4. STAB2–Software systems used to build classifiers with C4.5

Software System	Number of versions (major)	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
Jigsaw	4(3)	846–958
Jlex	4(2)	20–23
Lmjs	2(2)	106
Voji	4(4)	16–39

5.2. Algorithmic Settings

We performed two sets of experiments. In the first one, we used rule sets constructed by C4.5 to seed our GA. In the second set, we used randomly generated rule sets (the size of the random rule sets ranges between 5 and 70 rules per rule set and 2 to 30 conditions per rule). This allows us to test our approach with any rule sets not

Table 5. STAB2–Software systems used to train and test the GA

Software System	Number of versions (major)	Number of classes
Jedit	2(2)	464–468
Jetty	6(3)	229–285

Table 6. STAB2-Software quality metrics used as attributes in the classifiers

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
CUBF	number of classes used by a member function
Inheritance metrics	
NOC	number of children
NOP	number of parents
NON	number of nested classes
NOCONT	number of containing classes
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.
DEPCC	operation access metric
NPPM	number of public and protected meth. in a cl.
NPA	number of public attributes

just ones created with a machine learning technique. In both sets of experiments, the GA evolves the rule sets and adapts them to the in-house data (tables 3 and 5 The parameters to our GA had the following values[7]: We set crossover probability to 90%, mutation rate to 10%, percentage of chromosomes copied by elitism to 10% and number of generations through which the GA iterates to 50.

[7] Several experimentations with the algorithm showed that the parameter values have very little effect on the best accuracy obtained by this GA.

5.3. Results

To accurately estimate the accuracy of the rule sets, we used 10-fold cross-validation (10 is a commonly used number). In this technique, the data set is randomly split into 10 folds. Nine of them are used to train the GA and the remaining one to test it. The process is repeated for all 10 possible combinations. Also, in order to account for the element of randomness in the GA, we repeated each experiment 30 times and averages (and standard deviations) over the 30 runs are reported in tables 7 and 8.

The tables show the correctness and J_index on both the training and the testing sets of the best rule set constructed by C4.5, the best constructed by GA, the best constructed by GA when seeded with random rule sets (GA-R), the average of randomly generated rule sets (R) and the majority classifier (Maj.- the constant classifier that always predicts the majority classification label).

Since STAB1 is unbalanced, the majority classifier has a high accuracy while it performs poorly on the minority class. This makes it very important to assess the average correctness per classification label (J_index) on this data set. Table 7 shows that the GA significantly outperforms C4.5, random guess and the majority classifier in the J_index. It also shows that GA-R outperforms the three classifiers. It is important to point out that the objective function that the GA optimizes is the correctness of the rule set encoded by a chromosome. Hence, we believe that if we include the J_index in the fitness function, the improvement will be more significant.

In the case of STAB2, the GA slightly outperforms C4.5 and significantly random guess in both the correctness and the J_index. We believe that the slight improvement is due to the noisy nature of the data. Also, improving correctness on a balanced data set is more difficult than improving it on an unbalanced data set

Table 7. Results on STAB1. Correctness and J_index (standard deviation)

	$C_{testing}$	$C_{training}$	$j_{testing}$	$J_{training}$
C4.5	85(2)	85(0.5)	50(2)	50(0.5)
GA	85.5(3)	86(1)	59(4)	60.5(3)
GA-R	85(3)	85.5(1)	55.5(6.5)	56(6)
R	29.64(30.89)	29.71(30.84)	49.79(1.08)	50.15(0.27)
Maj.	85	85	42.5	42.5

Table 8. Results on STAB2. Correctness and J_index (standard deviation)

	$C_{testing}$	$C_{training}$	$j_{testing}$	$J_{training}$
C4.5	68(6)	68(0.5)	58(4)	58(0.5)
GA	70(6)	74.5(1)	60.5(5)	65(3)
GA-R	69(6)	73(3)	60(6)	65(4)
R	28(30.27)	30(28.23)	50.11(0.21)	50.12(0.21)
Maj.	62	62	31	31

Table 9. Complexity of rule sets: rules per rule set (conditions per rule)

	STAB1	STAB2
C4.5	2–7(1–5)	2–11(1–6)
GA	1–3(1–2)	1–5(1–3)
GA-R	2–5(1–4)	2–7(1–4)

[20]. The low accuracies in general, indicate that the problem of predicting stability of software components is a difficult one.

One important quality of our GA is the low complexity of the rule sets that it constructs. These have fewer attributes and conditions per rule than those that C4.5 builds (Table 9) which makes them easier to interpret by human experts. Another interesting observation is the speed with which the GA finds the "best" rule set (within the first 10 to 15 generations when seeded with C4.5 rule sets, 35 or later when seeded with random rule sets[8]). This explains the small number of generations mentioned earlier (50).

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed an adaptive approach to optimize existing software component quality estimation models. The approach starts with several rule sets and searches for a combination of rules, derived from them, that makes better rule sets with respect to a new data set. In real applications, this can be seen as taking rule sets built from common domain knowledge and finding a combination of their "expertise" that results in a rule set with a higher prediction accuracy when used on context specific data. The approach is derived from the genetic algorithm methodology. We have conducted experiments with two different data sets involving the stability of classes in an object-oriented system. One of the two stability data sets is unbalanced. The GA outperforms C4.5, random guess and the majority classifier significantly on the unbalanced data set but only slightly on the balanced one. The GA outperforms C4.5 when seeded with random rule sets. Our approach is independent of the software quality characteristic that is being estimated and hence it is interesting to see how well it performs on other characteristics (such as maintainability, reusability, etc.) especially with data containing less noise. The end result of our GA is a rule-based classifier. This provides two separate utilities: the estimation of a quality characteristic (stability in this case) and guidelines that can help software engineers to attain it. Due to the elitism operator, our technique guarantees no damage to results. In most cases, the rule set found by the GA is different from the initial one and hence, can be used as an additional guideline

[8] The difference is due to the size of the initial rule sets. Random rule sets are bigger in size than C4.5 rule sets so this provides a bigger search space.

during the development phase of a software product. In all cases, our approach results in rule sets that have a lower complexity than those constructed by C4.5. Future work involves training the GA on the J_index and comparing it to another heuristic (such as simulated annealing). Another interesting modification would be in the design of the genetic operators in a way that modifies the attributes in the conditions (not only the values) and also taking the gain factor into consideration in the objective function.

7. ACKNOWLEDGMENT

The first author's research was partially supported by a grant from CNRS-Lebanon (Conseil National de La Recherche Scientifique) and from the University Research Council at the Lebanese American University.

REFERENCES

1. G. M. Barnes and B. R. Swim, "Inheriting software metrics," *JOOP*, vol. 6, no. 7, pp. 27–34, Nov./Dec. 1993.
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. S. Chidamber and C. Kemerer, "A metrics suite for object-oriented design," in *IEEE Transactions on Software Engineering*, 1994, vol. 20, pp. 476–493.
4. J. C. Coppick and T. J. Cheatham, "Software metrics for object-oriented systems," in *CSC '92 Proceedings*, 1992, pp. 317–322.
5. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
6. B. Henderson-Sellers, "Some metrics for object-oriented software engineering," in *TOOLS Proceedings*, 1991, pp. 131–139.
7. W. Li and S. Henry, "Maintenance metrics for the object-oriented paradigm," in *Proceedings of the First International Software Metrics Symposium*, Baltimore, Maryland, 1993, pp. 52–60.
8. W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Systems Software*, vol. 23, pp. 111–122, 1993.
9. M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1994.
10. G. D.E. Rumelhart, Hinton, and R. Williams, "Learning representations by back-propagation errors," *Nature*, vol. 323, 1986.
11. J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
12. J. Holland, *Adaptation in Natural and Artificial Systems: an introductory analysis with applications to biology, control, and artificial intelligence*. Ann Arbor, MI: University of Michigan Press, 1975.
13. C. Darwin, *The Origin of Species*. John Murray, 1859.
14. U. Fayyad, "On the induction of decision trees for multiple concept learning," Ph.D. dissertation, EECS Department, University of Michigan, 1991.
15. S. Bouktif, D. Azar, H. Sahraoui, B. Kégl, and D. Precup, "Improving rule set-based software quality prediction: A genetic algorithm-based approach," *Journal of Object Technology*, vol. 3, no. 4, April 2004.
16. D. Azar, S. Bouktif, B. Kégl, H. Sahraoui, and D. Precup, "Combining and adapting software quality predictive models by genetic algorithms," in *Automated Software Engineering*, 2002.
17. L. Briand, J. Wüst, J. Daly, and V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, 2000.

18. L. Briand and J. Wust, “Empirical studies of quality models in object-oriented systems.” in *Advances in Computers*, M. Zelkowitz, Ed., 2000, vol. 20.
19. H. Zuse, *A Framework of Software emeasurement*. Walter de Gruyter, 1998.
20. T. Elomaa, “In defense of c4.5: Notes on learning one-level decision trees,” in *Machine Lea-69rning: Proceedings of the 11th International Conference*. Morgan Kaufmann, 1994, p. 62.