Paper: jc10-3-2514:2006/2/15

MASP – An Enhanced Model of Fault Type Identification in Object-Oriented Software Engineering

Atchara Mahaweerawat*, Peraphon Sophatsathit*, Chidchanok Lursinsap*, and Petr Musilek**

*Advanced Virtual and Intelligent Computing Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University
Phyathai Road, Patumwan, Bangkok 10330, Thailand
E-mail: atchara.m@student.chula.ac.th, {peraphon.s, lchidcha}@chula.ac.th

**Facility for Advanced Computational Intelligence and Applications (FACIA), Department of Electrical and Computer Engineering,
Faculty of Engineering, University of Alberta
W2-030 ECERF, Edmonton, Alberta T6G 2V4, Canada
E-mail: Petr.Musilek@ualberta.ca

[Received 00/00/00; accepted 00/00/00]

To remain competitive in the dynamic world of software development, organizations must optimize the use of their limited resources to deliver quality products on time and within budget. This requires prevention of fault introduction and quick discovery and repair of residual faults.
In this paper, a new model for predicting and identifying of faults in object-oriented software systems is introduced. In particular, faults due to the use of inheritance and polymorphism are considered as they account for significant portion of faults in object-oriented systems.
The proposed MASP model acts as a fault metric selector that gathers relevant filtering metrics suitable for specific fault types employing coarse-grained and fine-grained metric selection algorithms. A fault predictor is subsequently established to identify the fault type of individual fault classification.
It is concluded that the proposed model yields high discrimination accuracy between faulty and fault-free classes.

Keywords: software fault, predictive model, neural networks, fault metrics, fault prediction and identification

1. Introduction

Software reliability can be defined as the probability of failure-free operation of a computer program executing in a specified environment for a specified time [19]. It is often considered a software quality factor that can aid in predicting the overall quality of a software system using standard predictive models. Predictive models of software faults use historical and current development data to make predictions about faultiness of software subsystems/modules. Although software faults have been widely studied in both procedural and object-oriented programs, there are still many aspects of faults that remain unclear. This is true especially for object-oriented software systems in which inheritance and polymorphism can cause a number of anomalies and fault types [20]. Unfortunately, existing techniques used to predict faults in procedural software are not generally applicable in object-oriented systems.

Some recent studies [5, 6, 9, 11, 13, 17, 21] report the use of object-oriented metrics to predict fault-proneness and number of faults by applying various statistical methods and neural network techniques. However, they generally stop at the problem of fault prediction without attempting to further characterize the faults that are likely to present in the system. In this paper, a new method of fault prediction and fault type identification is introduced. For the reasons mentioned earlier, faults due to inheritance and polymorphism are of special interest in this work.

The problem of predicting whether a software class is faulty can be viewed as a binary classification problem in which the class represents a data point with coordinates described by object-oriented metrics and other parameters. The prediction of fault type in a faulty software class is then considered as a clustering problem in which each fault type is represented by a cluster prototype [8]. To solve the two problems, use of neural network techniques [12] is proposed. The classification problem is addressed using a Multilayer Perceptron (MLP), while the solution to clustering problem is derived based on Radial-Basis Function Network (RBFN).

The paper is organized as follows. Section 2 provides background information on the problem area including fault taxonomy, software metrics, and neural network methods used in this study. Section 3 describes a coarse-grained fault metric selector that serves as fault-prediction preprocessor. Besides the coarse-grained selected fault metrics, additional relevant metrics are extracted by a fine-grained fault metric selector in Section 4. Construction of the fault identification model employing the fine-grained selected metrics and the results are also presented. The results obtained from the model are further discussed in Section 5. Some related works pertaining to the proposed approach are also given in Section 6.

Table 1. Fault and anomalies due to inheritance and polymorphism.

Acronym	Fault/Anomaly
SDA	State Definition Anomaly
SDIH	State Definition Inconsistency
SDI	State Definition Incorrectly
IISD	Indirect Inconsistent State Definition
SVA	State Visibility Anomaly

Table 2. Syntactic inheritance patterns.

Acronym	Syntactic Pattern
ECE	Extension method Calls another Extension method
ECI	Extension method Calls Inherited methods
ECR	Extension method Calls Refining method
EDIV	Extension method Defines Inherited state Variable
RCE	Refining method Calls Extension method
RCI	Refining method Calls other Inherited method
RCR	Refining method Calls another Refining method
RCOM	Refining method Calls Overridden Method
RDIV	Refining method Defines Inherited state Variable
RUIV	Refining method Uses Inherited state Variable

Table 3. Software metrics from a software tool.

Software Metrics
AvgCyclomatic
AvgCyclomaticModified
AvgCyclomaticStrict
AvgLine
AvgLineCode Average line code
CountClassBase
CountClassCoupled (CBO)
CountClassDerived (NOC)
CountDeclClass
CountDeclInstanceMethod (NIM)
CountDeclInstanceVariable (NIV)
CountDeclInstanceVariablePrivate
CountDeclInstanceVariableProtected
CountDeclInstanceVariablePublic
CountDeclMethod (WMC)
CountDeclMethodAll (RFC)
CountDeclMethodFriend
CountDeclMethodPrivate
CountDeclMethodProtected
CountDeclMethodPublic
CountLine
CountLineCode
MaxCyclomatic
MaxCyclomaticModified
MaxCyclomaticStrict
MaxInheritanceTree (DIT)
PercentLackOfCohesion (LCOM)
Number of Parents(NOP)
Number of Direct Base classes(DirBase)
Number of Indirect Base Classes (IndBase)
Number of Descendants (NOD)

Finally, the main conclusion and direction of future work are given in Section 7.

2. Background

2.1. Neural Networks

This study employs two neural network techniques as the underlying mechanisms for fault prediction, namely, Multilayer Perceptron (MLP) and Radial-Basis Function Networks (RBFN). The former helps cluster input data into appropriate fault categories, whereas the latter computes the fault types so obtained via curve-fitting approximation. Their procedural details can be found in [12].

Table 4. Fault/anomaly types identified by syntactic patterns and parameters.

Pattern/ Parameter	Fault Type				
	SDA	SDIH	SDI	IISD	SVA
ECE	X				
ECI	X				X
ECR	X				
EDIV	X		X	X	
RCE	X		X	X	
RCI	X		X		X
RCR	X		X		
RCOM	X		X		
RDIV	X	X	X		
RUIV		X			
NMI	X	X	X		X
NME	X		X	X	X
NMR	X	X	X	X	X
DepIV	X				
DiffOvrrI			X		
DiffDef			X		
NDTRAM	X		X		
NDVRAM	X		X		
NDTRM	X		X		
NDVRM	X		X		
OVrrMet	X		X		
NTIMet	X		X		
NVIMet	X		X		
NTOVrrMet	X		X		
NVOVrrMet	X		X		
IdenVar		X			
ImRef		X			
IPriV					X
RIpriV					X

2.2. Fault Categories and Software Metrics

Inheritance and polymorphism provide many benefits in creativity, efficiency, and reuse of object-oriented software development. However, they can cause a number of anomalies and faults [20]. This study focuses on five fault types incurred by the use of polymorphism shown in **Table 1**.

A number of parametric measurements are introduced as faulty causes, i.e., number of appearances of syntactic fault pattern [3], and syntactic and structural measures. The metrics are summarized in **Tables 2**, **3**. Further details of each metric can be found in [1, 3, 7]. The parametric measurements are categorized according to the above five fault types shown in **Table 4**.

Besides the number of appearances of the patterns and software metrics described in [3, 7], additional parameters are defined for this study as follows:

- **Number of inherited methods (NMI):** This parameter can be used with ECI, EDIV, RCI, RDIV, and RUIV [3] patterns to detect SDA, SDI, SDIH, and SVA faults.
- **Number of extension methods (NME):** This parameter can be used with ECE, ECI, ECR, EDIV, and RCE [3] patterns to detect SDA, SDI, IISD, and SVA faults.
- **Number of refining methods (NMR):** This parameter can be used with ECR, RCE, RCI, RCR, RCOM,

RDIV, and RUIV [3] patterns to detect SDA, SDI, SDIH, IISD, and SVA faults.

- **Number of methods dependent on the inherited variable which is defined in the descendant class (DepIV):** This parameter can be used with ECE, ECI, ECR, EDIV, and RCE [3] patterns to detect faults of SDA type. If an extension method defines a state variable *v* and there is another method that depends on *v*, then an SDA exists.

- **Number of portions that the inherited variable is defined differently in the inherited method from the overridden method in the indirect base class (DiffOvrrI):** This parameter can be used with the RCI [3] pattern to detect faults of SDI type. If a refining method calls an inherited method *i* instead of the overridden method *o* and *i* defines the same state variable as those in the overridden method but the result of definition is different, then an SDI fault appears.

- **Number of portions that the inherited variable is defined differently from the ancestor (DiffDef):** This parameter can aid EDIV, RCE, RCR, RDIV, and RCOM [3] patterns to detect an SDI fault. If an extension method *e* or a refining method *r* defines an inherited variable in the manner that is different from the ancestor, then an SDI fault occurs. If a refining method calls the extension method *e* or the refining method *r*, an SDI fault also appears.

Comparisons between state variables in the ancestor class and those in the descendant class are as follows:

- **Number of variable types defined in the ancestor method which is refined in the descendant class (NDTRAM)**

- **Number of variables defined in the ancestor method which is refined in the descendant class (NDVRAM)**

- **Number of variable types defined in the refining method of the descendant class (NDTRM)**

- **Number of variables defined in the refining method of the descendant class (NDVRM)**

All four parameters above can be used with ECR, RCR, RDIV, and RCOM [3] patterns to detect the SDA and SDI faults. If a refining method *r* does not define the same set of state variables as in the ancestor class, an SDA fault appears. An SDA fault also exists if an extension method or another refining method calls the refining method *r*. Moreover, if the refining method *r* calls an overridden method *o* and defines additional state variables not defined by *o*, the SDA fault will be introduced.

If *r* defines the same set of state variables as overridden method *o* does but the definition is different, then an SDI fault occurs.

- **Number of overridden methods of the indirect base class (OvrrMet):** This parameter can help the RCI [3] pattern detect the faults of SDA and SDI types when the inherited methods are called instead of the overridden methods.

Comparisons between state variables in the inherited methods and those in the overridden methods are as follows:

- **Number of variable types defined in the inherited method which are called instead of the overridden method (NTIMet)**

- **Number of variable types defined in the overridden method of the indirect base class (NTOVrrMet)**

- **Number of variables defined in the inherited method which is called instead of the overridden method (NVIMet)**

- **Number of variables defined in the overridden method of the indirect base class (NVOVrrMet)**

All four parameters above can be used with the RCI [3] pattern to detect the faults of SDA and SDI types. If a refining method *r* calls an inherited method *i* instead of an overridden method *o* and the method *i* does not define the same set of state variables as in the method *o*, then an SDA fault exists.

However, if *i* defines the same set of state variables as *o* does but the definition is different, an SDI fault appears.

- **Number of identical name variables (IdenVar):** This parameter can be used with RDIV and RUIV [3] patterns to detect an SDIH fault.

- **Number of implicit references of the identical name variable (ImRef):** This parameter can be used with RDIV and RUIV [3] patterns and the parameter IdenVar to detect an SDIH fault. If there is a state variable *v* whose name is identical to one that is inherited and that is defined by a refining method, an SDIH fault exists. An SDIH fault will also occur if a refining method uses *v* to define an inherited state variable with the implicit reference.

- **Number of called inherited methods that define or use private variable (IPriV):** This parameter can be used with the RCI [3] pattern to detect an SVA fault. If a refining method *r* calls an inherited method *i* to modify a state variable which is declared private in the indirect base class, an SVA fault is likely to occur.

- **Number of refining methods in the ancestor class that are inherited to the descendant class (RIpriV):** This parameter can be used with the RCI [3] pattern and the parameter IPriV to detect an SVA fault. If a refining method *r* calls an inherited method *i* to modify a state variable which is declared private in an indirect base class and the method of the direct base class which inherited *i* is refined and

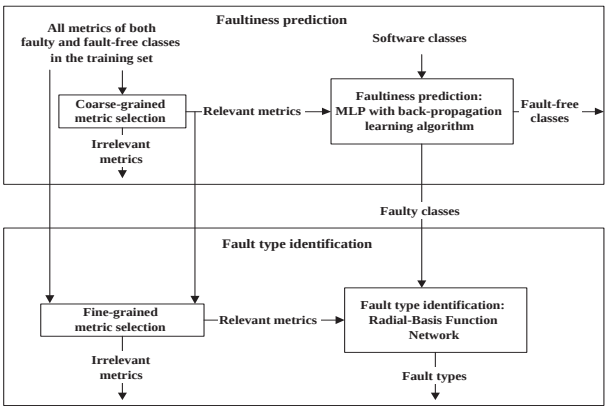

Fig. 1. Diagram of MASP fault identification model construction.

not consistent with the original method in the indirect base class, then an SVA fault appears.

2.3. Fault Analysis

In this study, a set of source code is examined to analyze faults that exist in software systems. Fault analysis consists of two parts; faultiness prediction and fault type identification.

A faultiness predictive model has been constructed based on software characteristics to predict whether the considered software is faulty or fault-free. A set of predetermined software metrics are used as the principal characterization attributes of software, while neural network techniques are applied to build the predictive model. In our previous work [16], two faultiness predictive models were built based on eleven software metrics with the help of multilayer perceptron (MLP) for the first model and Radial-basis function network (RBFN) for the second model. The results yielded prediction accuracy of 60% and 83%, respectively. Since some software metrics used in prior work are suitable only for structured software, additional object-oriented software metrics have been employed. A fault identification model named MASP is introduced. The MASP model consists of two stages, namely, faultiness prediction (or coarse-grained) stage and fault type identification (or fine-grained) stage. This is depicted in **Fig.1**.

In the faultiness prediction stage, a coarse-grained metric selection algorithm is proposed to extract the vital fault metrics that affect fault proneness. A faultiness predictive model is applied to extract faulty classes using multilayer perceptron with back-propagation learning algorithm.

Since the metrics selected by coarse-grained method do not contain adequate trace provisions for identifying fault type from the faulty classes so obtained, a fine-grained metric selection algorithm is presented to enhance trace identification capability with the help of other relevant metrics. A fault type identification model is constructed using radial-basis function network (RBFN). The MASP approach identifies not only fault type residing in the faulty classes, but also determines the degree of various impacts on which each fault type has. This is carried out by means of an algorithm which considers the metrics associating with the hidden nodes in the hidden layer of the model and their corresponding weights. Details on how the algorithm works will be elucidated in the sections that follow.

Table 5. Training and test data sets.

Fault Category	A		B		C	
	training	test	training	test	training	test
Fault-free	400	100	273	75	284	87
SDA	80	20	95	33	98	22
SDIH	80	20	116	14	91	21
SDI	80	20	91	31	126	25
IISD	80	20	102	27	112	19
SVA	80	20	123	20	89	26
Total	800	200	800	200	800	200

3. Faultiness Prediction – A Coarse-Grained Approach

The experiments have been carried out using 3,000 C++ classes from different sources: complete applications, individual algorithms, sample programs, and various other sources on the Internet. The classes were written by different developers. The size of the classes varies between 100 and 500 lines of code. Such composition of experimental data provides a good mixture necessary for obtaining general predictive models.

Of all the 3,000 classes, half of them were representatives of faulty samples and the other half were fault-free samples. The faulty samples were divided into five groups of 300 classes, having each fault type code listed in **Table 1** inserted according to syntactic patterns in [3]. All faulty and fault-free samples were measured with 60 software metrics and fault parameters given in [1, 3].

The data were normalized to 0 and 1, and randomly grouped into three sets, namely, A, B, and C. Each group was divided into an 800-class training set and a 200-class test set. **Table 5** shows the number of software classes in each fault type per set.

All 60 software metrics and fault parameters were applied to the experimental data. However, not all software metrics and fault parameters contributed to faultiness of the software classes. Therefore, it was necessary to select only the relevant metrics and fault parameters in order to filter out the irrelevant ones. Some researches [5, 6, 9, 10, 22] employed univariate logistic regression analysis and Principal Component Analysis (PCA) as a preprocessing scheme to extract only suitable object-oriented metrics for predictive model construction. Because the statistical and mathematical methods are black box which cannot explain the reasoning behind the metric selection [14], a new algorithm to select the relevant attributes is proposed. In the following discussion, both software metrics and fault parameters are simply referred to as metrics.

1. Separate the training set into two sets, namely, fault-free set for fault-free classes and faulty set for faulty classes.

2. In the fault-free set, calculate the average value of each metric.

$$AvgNFM_i = \frac{\sum_{j=1}^{p} x_i^j}{p}$$

where $AvgNFM_i$ is the average value of metric i in the fault-free set, $i = \{1, 2, \ldots, m\}$, m is the number of metrics, $j = \{1, 2, \ldots, p\}$, p is the number of fault-free classes in the fault-free set, and x_i^j is the value of metric i of the fault-free class j.

3. In the faulty set, calculate the average value of each metric.

$$AvgFTM_i = \frac{\sum_{k=1}^{q} y_i^k}{q}$$

where $AvgFTM_i$ is the average value of metric i in the faulty set, $i = \{1, 2, \ldots, m\}$, m is the number of metrics, $k = \{1, 2, \ldots, q\}$, q is the number of faulty classes in the faulty set, and y_i^k is the value of metric i of the faulty class k.

4. Calculate the relative difference of the average value of each metric between the fault-free and faulty sets.

$$DiffAvgM_i = \frac{|AvgNFM_i - AvgFTM_i|}{(AvgNFM_i + AvgFTM_i)} \times 100$$

where $DiffAvgM_i$ is the relative difference of the average value of metric i between the fault-free and faulty sets.

5. Select the metrics having the average relative difference above the selected threshold.

Applying the above selection algorithm using the threshold value of 50 to the training set A, eleven metrics were obtained.

There are feature selection techniques used in [5, 6, 9, 10, 22], i.e., univariate logistic regression, multivariate logistic regression, Principal Component Analysis (PCA), and an unsupervised method presented in [18]. We compared the performance of our model with other approaches. Different techniques were applied to find a subset of proper metrics, including the above pre-selected metrics, for faultiness predictive model using MLP with back-propagation learning algorithm based on the above pre-selected metrics. The objective of the models is to correctly classify the data points into fault-free and fault groups. The structure of each faultiness predictive model consists of input nodes with respect to the selected metrics in the input layer, 15 hidden nodes in the hidden layer, and 1 output node in the output layer.

The expected output value computed from the output node of each model would be zero for the fault-free class and one for the faulty class. The actual output was carried out during the training process. Each output value was computed from sigmoid function in batch mode using a 0.35 learning rate value, along with the adjusted weights (in accordance with the delta rule without a momentum term), and input values. The training process terminated when the error was less than 0.001 or reached 1000 epoches. The output values so obtained ranging between 0 and 1 were indecisive for data classification. Setting an acceptance ratio at 0.55, a data point could be classified as a faulty class if the output of MLP was greater than this value. Otherwise, it would be a fault-free class. The comparative results of the experiments are depicted in **Table 6**. Each faultiness predictive model was built from the training set A and re-applied to the test set A, data set B and C. The experiments were carried out on Matlab V6.0. Three models applying the sets of metrics obtained from univariate logistic regression, multivariate logistic regression, and the proposed coarse-grained algorithm are shown in **Table 7**.

Table 6. Results from faultiness predictive models based on sets of metrics obtained from different metric selection techniques.

Metric selection technique	Test set		
	A	B	C
Univariate logistic regression	90.00%	87.20%	88.00%
Multivariate logistic regression	92.50%	88.10%	88.20%
Principal component analysis	77.50%	74.60%	72.50%
An unsupervised method [Mitra]	68.50%	70.40%	67.20%
Random selection	76.00%	75.10%	74.40%
The proposed coarse-grained algorithm	95.50%	94.90%	95.40%

Table 7. The selected metrics obtained from applying univariate logistic regression, multivariate logistic regression, and the proposed coarse-grained algorithm to training set A.

Univariate logistic regression	Multivariate logistic regression	The proposed coarse-grained algorithm
ImRef	NOD	NOC
DiffDef	ImRef	CountDeclInstanceVariableProtected
DiffOVrrI	DiffDef	CountDeclMethodProtected
RIpriV	DepIV	NOD
	RIpriV	ECE
		ECR
		ImRef
		DiffDeff
		DiffOVrrI
		DepIV
		RIpriV

The highest correctness percentage was accomplished by our model and was subsequently evaluated through some measurement criteria [15] as follows:

- **Type 1 error (T1):** This error occurs when a faulty class is classified as fault-free; T1 = 2.81%
- **Type 2 error (T2):** This error occurs when a fault-free class is classified as faulty; T2 = 2%
- **Quality achieved (C):** If all faulty classes are properly classified, defects will be removed by extra ver-

ification; C = 95.51%

- **Inspection (I):** Inspection measures the overall verification cost by considering the percentage of classes that should be verified; I = 61.95%
- **Waste Inspection (WI):** Waste inspection is the percentage of classes that do not contain faults but are verified because they have been classified incorrectly; WI = 3.23%

4. Fault Type Identification – A Fine-Grained Approach

A fine-grained metric selection algorithm has been proposed. The algorithm is based on the relative difference between the value of each metric applied to faulty and fault-free classes in the training set.

1. Set initial weight of each metric to accentuate its importance.

$$W_i^{(t)} = 0$$

where $W_i^{(t)}$ is the weight value of metric i at iteration t,$i = \{1,2,\ldots,m\}$, m is the number of metrics, and t is the iteration number.

2. Establish a pair of fault-free and faulty classes from the training set, each of which consists of the same corresponding set of metrics.

$$X = \{x_1, x_2, \ldots, x_m\}, Y = \{y_1, y_2, \ldots, y_m\}$$

where X is a faulty class consisting of m metrics, Y is a fault-free class consisting of m metrics, x_i is the value of metric i of the faulty class, and y_i is the value of metric i of the fault-free class.

3. Calculate the relative difference of each metric pair from step 2.

$$D_i = \frac{|x_i - y_i|}{(x_i + y_i)} \times 100$$

where D_i is the relative difference of metric i among their respective classes, x_i is the value of metric i of the faulty class, y_i is the value of metric i of the fault-free class. This will prevent metric intermixing among their corresponding applicable domains.

4. Adjust the weight value of each metric according to the following conditions:

$$IF\ D_i \geq \beta\ THEN\ W_i^{(t)} = W_i^{(t-1)} + 1$$
$$ELSE\ W_i^{(t)} = W_i^{(t-1)} - 1$$

where $\beta = 50$ (in percentage) is a predefined threshold value.

Table 8. The combined filtered metrics.

Metrics from coarse-grained algorithm	Metrics from fine-grained algorithm
NOC	RUIV
CountDeclInstanceVariableProtected	NDTRAM
CountDeclMethodProtected	NDVRM
NOD	CountDeclInstanceVariablePublic
ECE	CountDeclMethodPrivate
ECR	OVrrMet
ImRef	CountDeclInstanceVariablePrivate
DiffDef	NDVRAM
DiffOvrrI	ECI
DepIV	RCOM
RIpriV	NDTRM
	CBO
	IndBase
	IdenVar
	EDIV
	NTIMet
	RCE
	NVIMet
	RCI
	NTOVrrMet
	RCR
	NVOVrrMet
	RDIV
	IPriV

5. Repeat step 2 through step 4 until all fault-free classes match with all faulty classes of the training set.

6. Consider the weight value of each metric, replacing negative values with zero

$$IF\ W_i < 0\ THEN\ W_i = 0$$

7. Normalize all weight values

$$W_i = \frac{W_i - min}{max - min}$$

where *max* and *min* are the maximum and minimum weight values, respectively.

8. Select the metrics with weight values above the selected threshold.

After applying the selection algorithm using the threshold value of 0.5, thirty-four relevant metrics were obtained from set A, B, and C. The metric union of all three sets yielded a combined 35 metrics, where all metrics from set A and C were identical, but B differed by only one. Note in **Table 8** that the thirty-five fine-grained selected metrics were composed of the same eleven metrics obtained from the coarse-grained algorithm in Section 3 and the newly added twenty-four metrics.

The construction of fault type identification model is based on RBFN technique and the fine-grained selected metrics as mentioned earlier. The model consists of 35 input nodes in the input layer, a number of hidden nodes in the hidden layer (this number is determined during the training process), and five output nodes in the output layer that form an output vector. The output vector denotes the type of fault in binary format as '10000', '01000',

Table 9. Results from applying the fault type predictive model to predict faulty classes.

Fault Category	Predicted Fault Type				
	SDA	SDIH	SDI	IISD	SVA
SDA	180	3	17	3	3
SDIH	5	246	6	2	3
SDI	20	6	261	4	2
IISD	13	4	11	243	9
SVA	5	2	2	5	264
Fault-Free	40	0	0	0	4

'00100', '00010', and '00001', representing SDIH, IISD, SVA, SDA, and SDI faults, respectively.

During the experiment, training data were used to generate the weights between the hidden layer and the output layer. If the network yielded low accuracy, the number of hidden nodes would be incremented by one. This restructuring by node-plus-one progression continued until the desired accuracy was acquired or the number of hidden nodes reached the number of training data points.

Based on the above procedures, the proposed model yielded a 91.38% prediction accuracy on faulty classes of test data from set A, all data from data sets B and C. The model was reapplied to the predicted faulty classes obtained from the faultiness predictive model and yielded the prediction accuracy of 87.60%. The reason behind the lower accuracy was that some fault-free classes were incorrectly classified as faulty classes by the faultiness predictive model in Section 4. The results shown in **Table 9** relate the actual number of each fault type and classification. Note that the effects of erroneous prediction become apparent as the fault-free classes are inferred to have SDA and SVA faults. Such caveats will impede future identification of the occurrence of these two fault types.

From the structure of the model, weights are assigned to the hidden layer and the output layer of fault type model. The weight value of each hidden node designates on which output node it would have an effect. The maximum weight value obtained from all hidden nodes that exert on a given output node indicates the dominance of the hidden node.

To explore which metrics dominate the fault type of a given hidden node that represents all 35 metrics, an algorithm is proposed as follows:

1. Choose a fault type to find a set of representative metrics.
2. Among the hidden nodes, find the one that has the most effect on fault type according to the weight values between the hidden nodes and output nodes.
3. Identify the set of classes from the training data where the selected fault is originated.
4. For each metric, calculate the difference between the metric values of a training class and a hidden node (each of which contains 35 metrics).

$$V_i^{(j,k)} = \left| c_i^k - h_i^j \right|$$

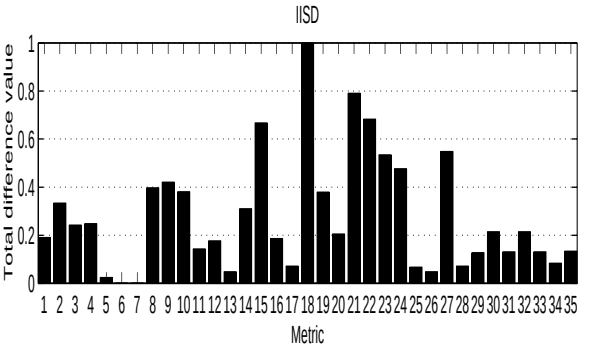

Fig. 2. The total difference of each metric between hidden nodes and training classes having IISD fault.

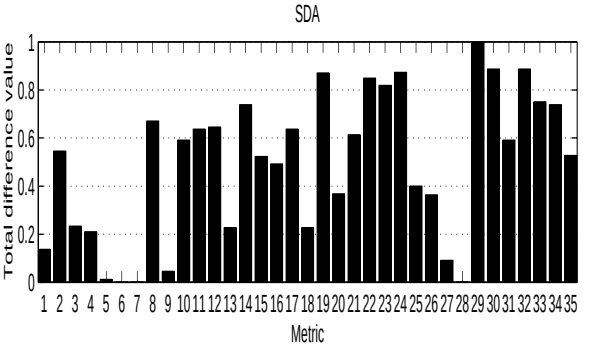

Fig. 3. The total difference of each metric between hidden nodes and training classes having SDA fault.

where $V_i^{(j,k)}$ is the difference of metric i among training class k and the hidden node j, c_i^k is the value of metric i of class k, and h_i^j is the value of metric i of hidden node j.

5. Repeat step 4 for the selected fault type until all classes and hidden nodes are considered.
6. For each fault type, calculate the total difference of each metric value from Step 5.

$$TotV_i = \sum_{j=1}^{m} \sum_{k=1}^{n} V_i^{(j,k)}$$

where $TotV_i$ is the total difference of metric i among all classes and hidden nodes, $V_i^{(j,k)}$ is the difference of metric i among training class k and hidden node j, m is the number of hidden nodes for the selected fault type, and n is the number of training classes for the selected fault type.

7. Normalize all total difference values by

$$TotV_i = \frac{TotV_i - min}{max - min}$$

where *max* and *min* are the maximum and minimum total difference values, respectively.

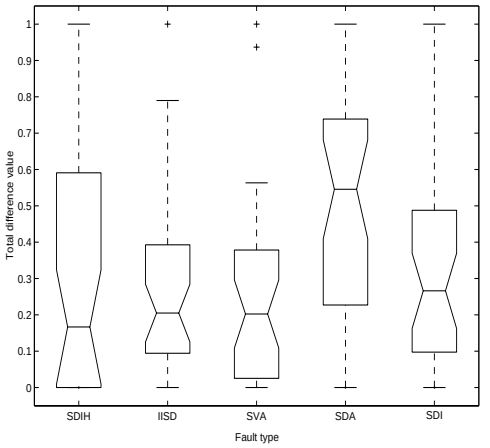

Fig. 4. The total difference of all metrics between hidden nodes and training classes for each fault type.

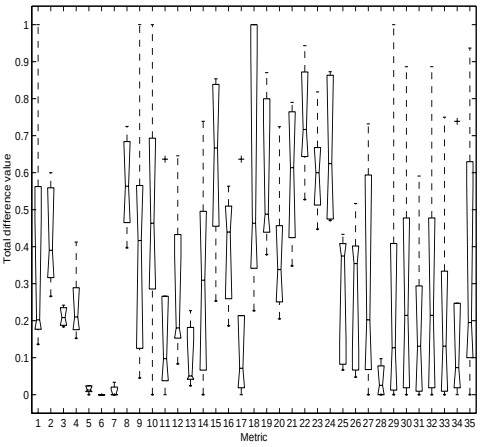

Fig. 5. The total difference of all metrics between hidden nodes and training classes for all fault types.

8. Repeat Steps 1-7 above until all fault types are considered.

Figures 2 and **3** show the effects of IISD and SDA metrics have on particular fault types. The zero total difference value means that the corresponding metrics of that training class and hidden node are the same and thus has no effect on the fault type. On the other hand, if the total difference metric between the training classes and the hidden nodes is high, that metric will likely contribute to the fault prediction of the software. As depicted in **Fig.3**, the 29^{th} metric represents the effect of SDA fault due to the number of variable types defined in the inherited method being called instead of the overridden method (NTIMet). In contrast, **Fig.2** shows that this metric has less effect on IISD fault.

Fig.4 demonstrates how important all metrics are in each fault type. There are many metrics affecting SDA fault with high scale of the total difference value, while other metrics affect IISD and SVA faults at low scale of the total difference value. The importance of each metric for all fault types is shown in **Fig.5**. Notice that the 18^{th} metric shows the highest effect of number of appearances of the pattern refining method (RDIV) [3] has on all fault types, while the 6^{th} metric depicts less effect of the number of private methods declared in a class (CountDeclMethodPrivate) [1] has on every fault type.

5. Discussion

The proposed coarse-grained software metric attribute selection algorithms of MASP proved to be effective in determining the significance of each metric and characterization of software faultiness. Based on the selected metrics and MLP with back-propagation learning algorithm, the proposed approach is able to predict faultiness of a class with more than 90% accuracy. According to the evaluation criteria, the faulty classes can be detected in 95.51% of test cases, the inspection cost for verification is 61.95%, and the waste cost is 3.23%. Only 2.81% of faulty classes are undetected.

Table 10. Results of fault type identification model obtained from the coarse-grained and fine-grained selected metric sets.

Test set	Metric set			
	Coarse-grained selected metrics		Fine-grained selected metrics	
	faulty classes	predicted faulty classes	faulty classes	predicted faulty classes
A	87.00%	81.55%	92.00%	86.41%
B	88.80%	85.60%	90.59%	89.20%
C	83.46%	80.19%	91.09%	86.15%

The proposed MASP's coarse-grained metric selection demonstrates slight advantages of fault-metric classification over conventional statistical and PCA approaches. However, only the coarse-grained selected fault metrics were not enough for fault type identification, a fine-grained metric selection algorithm was proposed to further extract additional relevant metrics that affect the corresponding fault type. Such preprocessing ground work establishes an effective filtering mechanism that permits higher accuracy of subsequent fault type identification as depicted in **Table 10**. The fault type predictive model applying the coarse-grained selected metrics yields an average of 85% and 82% accuracy on faulty classes and predicted faulty classes, respectively. In contrast, the predictive model obtained from the fine-grained metrics yields an average of 91% and 87% accuracy on faulty classes and predicted faulty classes, respectively. Moreover, the primary cause of the contributing fault types can also be identified by MASP's pair-wise metric comparison algorithm in Section 4. In so doing, this two-stage fault prediction technique offers not only high accuracy fault prediction outcomes, but also the corresponding fault types that contribute to the designated faults. We envision that some forms of fine grained metric preprocessing for each particular fault type should be carried out to alleviate the aforementioned caveats (as shown in **Tables 9** and **10**) and the costs incurred.

6. Related Works

Toshihiro Kamiya et al. [13] proposed a method to estimate the fault-proneness of the class in the early phase using several complexity metrics for object-oriented software and multivariate logistic regression analysis. They introduced four checkpoints into the analysis/design/implementation phase, and estimate the fault-prone classes using the applicable metrics at each checkpoint.

In [6], a fault-proneness prediction model was built based on a set of object-oriented measures using data collected from a mid-size Java system employing logistic regression analysis.

P. Kokol et al. [14] introduced some methods for reliability prediction based on software metrics, presented the results using these methods with a large database of modules in C language. The results and methods were compared. They found that statistical and mathematical methods accurately predicted the reliability of software modules, where black box methods could not explain the reasons behind the prediction.

In [4], neural networks were proposed as an alternative technique to build software reliability growth models. A comparison between regression parametric models and neural network models was carried out and concluded that neural networks were able to provide models with small Sum Square Error (SSE) than the regression model in all considered cases.

Mie Mie Thet Thwin and Tong-Seng Quah [21] presented the application of neural networks for predicting number of faults in three industrial real-time systems based on object-oriented design metrics. Ward Network which is a backpropagation network was applied to construct a neural network model. They concluded that neural network model could predict the number of faults more accurately than multiple regression model for software engineering data.

Khaled El Emam et al. [9] employed univariate logistic regression analysis for selecting some object-oriented design metrics. The proper metrics were applied with multivariate logistic analysis to construct a model for predicting which classes in a future release of a commercial Java application would be faulty.

Lionel C. Briand et al. [5] empirically explored the relationships between existing object-oriented coupling, cohesion, inheritance measures, and the probability of fault detection in system classes during testing. Principal component analysis and logistic regression were applied to select the proper metrics and built a prediction model.

D. Glasberg et al. [11] performed empirical study with the data from a commercial Java application using logistic regression technique. They found that Depth of Inheritance Tree (DIT) is a good measure of familiarity and has a quadratic relationship with fault-proneness.

Yida Mao, H.A. Sahroui, and Hakim Lounis [17] presented an experiment to verify three hypotheses about the impact of three internal characteristics (inheritance, coupling, and complexity) of object-oriented applications on reusability. The verification was done through a machine-learning approach and the experimental results showed that the selected metrics could predict with high level of accuracy on potentially reusable classes.

Ping Yu and Tarja Systä [22] empirically validated a set of object-oriented metrics in terms of their usefulness in predicting fault-proneness. Eight hypotheses on the correlations of the metrics with fault-proneness were given and tested on a system written in Java. Validation was statistically carried out using regression analysis and discriminant analysis.

F. Fioravanti and P. Nesi [10] analyzed more than 200 different object-oriented metrics extracted from the literature with the aim of identifying suitable models for detection of fault-proneness of classes. The work had been focused on identifying models that could detect as many faulty classes as possible and, at the same time, models that were based on a manageable small set of metrics. To reach their goal, principal component analysis was applied to find the subset of metrics and multivariate logistic regression analysis to construct the models.

Besides the prediction of fault-proneness in object-oriented software, fault type is also detected in [2]. Roger T. Alexander et al. defined a set of experiments, encompassing relative effectiveness of several coupling-based OO testing criteria and branch coverage. All OO testing criteria were more effective at detecting faults due to the use of inheritance and polymorphism than branch coverage.

7. Conclusion

The application of neural networks in predicting software faults requires enormous amounts of data. Analyzing the data is a major undertaking that must be carried out with the help of proper models. This study proposes a two-stage fault prediction model called MASP model. The first stage involves coarse-grained metric selection and faultiness prediction. The next stage performs fault identification by means of RBFN to categorize the faults according to several defined fault types based on the fine-grained selected metrics.

Some approaches have been explored to enhance the predictive model. The first possibility is to add more parameters. However, it is very difficult to find a proper set of parameters that can represent the characteristics of each fault type. Second, proper data and metrics classification techniques (preprocessing) enhance not only the efficiency of the training process, but also the performance of the predictive model in terms of precision. Accurate predictions obtained from such a good reliability model eventually lead to higher efficiency of software process and quality of resulting software products.

Acknowledgements

This research is financially supported by The Office of Higher Education Commission, Ministry of Education, Thailand.

References:

[1] "Understand for C++," Scientific Toolworks, Inc., St. George, Utah, http://www.scitools.com.

[2] R. T. Alexander, J. Offutt, and J. M. Bieman, "Fault Detection Capabilities of Coupling-based OO Testing," In Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), pp. 207-218, November, 2002.

[3] R. T. Alexander, J. Offutt, and J. M. Bieman, "Syntactic Fault Patterns in OO Programs," In Proceedings of the Eight International Conference on Engineering of Complex Computer Software, pp. 193-202, December, 2002.

[4] S. H. Aljahdali, A. Sheta, and D. Rine, "Prediction of Software Reliability: A Comparison between Regression and Neural Network Non-Parametric Models," In Proceedings of IEEE International Conference on Computer Systems and Applications (AICCSA'01), pp. 470-473, June, 2001.

[5] L. Briand, J. Wüst, and J. W. Daly, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," Journal of Systems and Software, 51, pp. 245-273, 2000.

[6] L. C. Briand, J. Wüst, and J. W. Daly, "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects," IEEE Transactions on Software Engineering, 28(7), pp. 706-720, 2002.

[7] S. R. Chidamber, and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, 20(6), pp. 476-493, 1994.

[8] P. Eklund, and L. Kallin, "Fuzzy Systems," Lecture Notes prepared for courses at the Department of Computing Science at Umeå University, Sweden, February, 2000.

[9] L. Emam, J. Wüst, and J. W. Daly, "The Prediction of Faulty classes Using Object-Oriented Design Metrics," Journal of Systems and Software, 56, pp. 63-75, 2001.

[10] F. Fioravanti, and P. Nesi, "A study on fault-proneness detection of Object-Oriented systems," In Proceedings of the fifth Conference on Software Maintenance and Reengineering (CSMR'01), pp. 121-130, March, 2001.

[11] D. Glasberg, and K. E. Emam, "Validating Object-Oriented Design Metrics on a Commercial Java Application," Technical Report NRC/ERB-1080, September, 2000.

[12] S. Haykin, "Neural Networks," Prentice Hall, the United States of America, 1999.

[13] T. Kamiya, S. Kusumoto, and K. Inoue, "Prediction of Fault-Proneness at Early Phase in Object-Oriented Development," In Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 253-258, May, 1999.

[14] P. Kokol, V. Podgorelec, M. Zorman, and M. Šprogar, "An Analysis of Software Correctness Prediction Methods," In Proceedings of the Second Asia-Pacific Conference on Quality Software (APAQS'01), pp. 33-39, December, 2001.

[15] F. Lanubile, "Evaluating Predictive Models Derived From Software Measure," Journal of Systems and Software, 38(1), pp. 225-234, 1996.

[16] A. Mahaweerawat, P. Sophatsathit, and C. Lursinsap, "Software Fault Prediction Using Fuzzy Clustering and Radial-Basis Function Network," In Proceedings of the International Conference on Intelligent Technologies, InTech/VJFuzzy'2002, pp. 304-313, December, 2002.

[17] Y. Mao, H. A. Sahraoui, and H. Lounis, "Reusability Hypothesis Verification Using Machine Learning Techniques: a case study," In Proceedings of the 13th IEEE International Conference on Automated Software Engineering, pp. 84-93, October, 1998.

[18] P. Mitra, C. Murthy, and S. K. Pal, "Unsupervised Feature Selection Using Feature Similarity," IEEE Transactions on Pattern Analysis and Machine Intelligence, 24(3), pp. 301-312, 2002.

[19] J. D. Musa, A. Iannino, and K. Okumoto, "Software Reliability Measurement, Prediction, Application," McGraw-Hill Book Company, the United States of America, 1987.

[20] J. Offutt, and R. Alexander, "A Fault Model for Subtype Inheritance and Polymorphism," In 12th International Symposium on Software Reliability Engineering, pp. 84-95, November, 2001.

[21] M. M. T. Thwin, and T.-S. Quah, "Application of Neural Network for Predicting Software Development Faults Using Object-Oriented Design Metrics," In Proceedings of the 9th International Conference on Neural Information Processing, pp. 2312-2316, November, 2002.

[22] P. Yu, and T. Systa, "Predicting Fault-Proneness using OO Metrics An Industrial Case Study," In Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR'02), pp. 99-107, March, 2002.

Name:
Atchara Mahaweerawat

Affiliation:
Advanced Virtual and Intelligent Computing Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University

Address:
Phyathai Road, Patumwan, Bangkok 10330, Thailand

Brief Biographical History:
2000-present Advanced Virtual and Intelligent Computing Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University
2004 Visiting Student, Department of Electrical and Computer Engineering, University of Alberta, Canada

Main Works:
- A. Mahaweerawat, P. Sophatsathit, and C. Lursinsap, "Software Fault Prediction Using Fuzzy Clustering and Radial-Basis Function Network," In Proceedings of the International Conference on Intelligent Technologies, InTech/VJFuzzy'2002, pp. 304-313, December, 2002.
- A. Mahaweerawat, P. Sophatsathit, C. Lursinsap, and P. Musilek, "Fault Prediction in Object-Oriented Software Using Neural Network Techniques," In Proceedings of the International Conference on Intelligent Technologies 2004, InTech'04, pp. 27-34, December, 2004.

Name:
Peraphon Sophatsathit

Affiliation:
Advanced Virtual and Intelligent Computing Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University

Address:
Phyathai Road, Patumwan, Bangkok 10330, Thailand

Brief Biographical History:
1994-1997 National Electronics and Computer Technology Center (NECTEC), Bangkok, Thailand
1997-present Department of Mathematics, Faculty of Science, Chulalongkorn University

Main Works:
- N. Arch-int, P. Sophatsathit, and Y. Li, "Ontology-Based Metadata Dictionary for Integrating Heterogeneous Information Sources on WWW," Journal of Research and Practice in Information Technology, Vol.35, No.4, pp. 285-302, November, 2003.
- S. Nakkrasae, and P. Sophatsathit, "An RPCL-Based Indexing Approach for Software Components Classification," International Journal of Software Engineering and Knowledge Engineering, Vol.14, No.5, pp. 497-518, October, 2004.
- S. Sai-ngern, C. Lursinsap, and P. Sophatsathit, "An address mapping approach for test data generation of dynamic linked structures," Information and Software Technology Journal, Vol.47, Issue 3, pp. 199-214, March 1, 2005.

Membership in Learned Societies:
- IEEE

Name:
Chidchanok Lursinsap

Affiliation:
Advanced Virtual and Intelligent Computing Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University

Address:
Phyathai Road, Patumwan, Bangkok 10330, Thailand

Brief Biographical History:
1978-1979 Department of Computer Engineering, Chulalongkorn University, Thailand
1982-1986 Department of Computer Science, University of Illinois at Urbana-Champaign, USA
1986-1995 Center for Advanced Computer Studies, University of Louisiana, USA
1995-present Department of Mathematics, Chulalongkorn University, Thailand

Main Works:
- R. Talumassawatdi, and C. Lursinsap, "Fault Immunization Concept for Self-Organizing Mapping Neural Networks," International Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems, Vol.9, No.6, pp. 781-790, 2001.
- K. Chinnasarn, C. Lursinsap, and V. Palade, "Blind Extraction of Mixed Kurtosis Signed Signal Using Partial Observations and Low Complexity Activation Functions," International Journal of Computational Intelligence and Applications, Vol.4, No.2, pp. 207-223, 2004.
- W. Wettayaprasit, C. Lursinsap, and C. H. Chu, "Extracting Linguistic Quantitative Rules From Supervised Neural Networks," International Journal of Knowledge-Based Engineering Systems, Vol.8, No.3, pp. 161-170, 2004.
- S. Sai-ngern, C. Lursinsap, and P. Sophatsathit, "An Address Mapping Approach for Test Data Generation of Dynamic Linked Structures," Information and Software Technology, 47, pp. 199-214, 2005.
- T. Raichareon, and C. Lursinsap, "A Divide-and-Conquer Approach to the Pairwise Opposite Class Nearest Neighbor (POC-NN) Algorithm," Pattern Recognition Letters, 26, pp. 1554-1557, 2005.

Membership in Learned Societies:
- IEEE
- Neural Network Society
- Sigma Xi
- Royal Institute of Thailand

Name:
Petr Musilek

Affiliation:
Facility for Advanced Computational Intelligence and Applications (FACIA), Department of Electrical and Computer Engineering, Faculty of Engineering, University of Alberta

Address:
W2-030 ECERF, Edmonton, Alberta T6G 2V4, Canada

Brief Biographical History:
1997- NATO Science Fellow (Intelligent Systems), University of Saskatchewan, Canada
1999- Joined University of Alberta, Canada
2005- Visiting Scientist, Institute of Computer Science, Academy of Sciences, Czech Republic
2005 - Visiting Professor, Department of Computer Science, University of Carlos III of Madrid, Spain

Main Works:
- P. Musilek, and M. M. Gupta, "Fuzzy Neural Models Based on Some New Fuzzy Arithmetic Operations," Journal of Advanced Computational Intelligence, Vol.3, No.4, pp. 245-254, 1999.
- P. Musilek, R. Guanlao, and G. Barreiro, "Genetic Programming of Fuzzy Aggregation Operations," International Journal of Intelligent and Fuzzy Systems, 16, pp. 107-118, 2005.
- P. Musilek, A. Lau, M. Reformat, and L. Wyard-Scott, "Immune programming," Information Sciences, 2006 (in press).

Membership in Learned Societies:
- Institute for Electrical and Computer Engineers (IEEE)
- North American Fuzzy Information Processing Society (NAFIPS)