การคาดคะเนข้อบกพร่องของซอฟต์แวร์โดยเทคนิคทางตรรกศาสตร์วิภัชนัยและโครงข่ายประสาทเทียม

นางสาวอัจฉรา มหาวีรวัฒน์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรดุษฎีบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์ ภาควิชาคณิตศาสตร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2549
ISBN 974-14-3911-3
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

SOFTWARE FAULT PREDICTION BY FUZZY LOGIC AND NEURAL NETWORK TECHNIQUES

Miss Atchara Mahaweerawat

A Dissertation Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy Program in Computer Science

Department of Mathematics

Faculty of Science

Chulalongkorn University

Academic year 2006

ISBN 974-14-3911-3

Thesis Title	SOFTWARE FAULT PREDICTION BY FUZZY LOGIC AND NEURAL NETWORK TECHNIQUES
By	Miss Atchara Mahaweerawat
Field of study	Computer Science
Thesis Advisor	Associate Professor Peraphon Sophatsathit, Ph.D.
Thesis Co-advisor	Professor Chidchanok Lursinsap, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctor's Degree

[illegible] .... Dean of the Faculty of Science
(Professor Piamsak Menasveta, Ph.D.)

THESIS COMMITTEE

[illegible] .... Chairman
(Associate Professor Jack Asavanant, Ph.D.)

[illegible] .... Thesis Advisor
(Associate Professor Peraphon Sophatsathit, Ph.D.)

[illegible] .... Thesis Co-advisor
(Professor Chidchanok Lursinsap, Ph.D.)

[illegible] .... Member
(Associate Professor Somchai Prasitjutrakul, Ph.D.)

[illegible] .... Member
(Assistant Professor Somjai Boonsiri, Ph.D.)

[illegible] .... Member
(Surapant Meknavin, Ph.D.)

อัจฉรา มหาวีรวัฒน์ : การคาดคะเนข้อบกพร่องของซอฟต์แวร์โดยเทคนิคทางตรรกศาสตร์วิภัชนัยและโครงข่ายประสาทเทียม. (SOFTWARE FAULT PREDICTION BY FUZZY LOGIC AND NEURAL NETWORK TECHNIQUES) อ. ที่ปรึกษา : รศ. ดร. พีระพนธ์ โสพัศสถิตย์, อ.ที่ปรึกษาร่วม : ศ. ดร. ชิดชนก เหลือสินทรัพย์, 180 หน้า. ISBN 974-14-3911-3.

ในโลกของการพัฒนาซอฟต์แวร์ องค์กรต่างๆจะต้องจัดการใช้ทรัพยากรที่มีอยู่อย่างจำกัดให้เหมาะสมเพื่อส่งมอบซอฟต์แวร์ที่มีคุณภาพภายใต้ระยะเวลาที่กำหนดและงบประมาณที่จำกัด ดังนั้นการค้นหา แก้ไข และป้องกันข้อบกพร่องที่เกิดขึ้นระหว่างการพัฒนาซอฟต์แวร์ หรือ ข้อบกพร่องที่ยังมีอยู่ในขั้นตอนการบำรุงรักษาซอฟต์แวร์จึงเป็นสิ่งจำเป็น การทดสอบซอฟต์แวร์เป็นวิธีที่ใช้กันทั่วไปเพื่อหาข้อบกพร่องของซอฟต์แวร์อย่างเป็นขั้นตอนกับชุดทดสอบที่เหมาะสม ซึ่งวิธีนี้จะเสียค่าใช้จ่ายสูงและใช้ความพยายามอย่างมาก วิทยานิพนธ์นี้ได้นำเสนอวิธีการคาดคะเนข้อบกพร่องของซอฟต์แวร์โดยไม่ต้องติดตั้งและใช้งานซอฟต์แวร์จริง กระบวนการคาดคะเนข้อบกพร่องของซอฟต์แวร์ประกอบด้วย 4 ขั้นตอนคือ การคาดคะเนความบกพร่อง การคาดคะเนประเภทข้อบกพร่อง การคาดคะเนข้อบกพร่องชนิดพลวัต และการหาตำแหน่งของข้อบกพร่อง โดยการสร้างแบบจำลองการคาดคะเนข้อบกพร่องสำหรับแต่ละขั้นตอนบนพื้นฐานของซอฟต์แวร์เมตริกซ์ด้วยเทคนิคทางตรรกะวิภัชนัยและโครงข่ายประสาทเทียม เมื่อระบุได้ว่าซอฟต์แวร์มีความบกพร่อง ซอฟต์แวร์ดังกล่าวจะถูกพิจารณาเพื่อหาตำแหน่งของข้อบกพร่องและผลของการทำนายความบกพร่องยังถูกวิเคราะห์ต่อเพื่อหาซอฟต์แวร์เมตริกซ์ที่มีความสำคัญต่อการทำนายความบกพร่องด้วยเทคนิคการวิเคราะห์ความไหว ดังนั้นวิธีการที่นำเสนอนี้จะเป็นพื้นฐานสำหรับการสร้างตัวแบบด้วยกลไกการเรียนรู้เพื่อการประกันคุณภาพของซอฟต์แวร์ อีกทั้งช่วยลดระยะเวลาที่ใช้ในการตรวจสอบซอฟต์แวร์ที่มีความผิดพลาดง่ายเพื่อให้ได้มาซึ่งซอฟต์แวร์ที่มีคุณภาพและไว้ใจได้

ภาควิชา....คณิตศาสตร์............................	ลายมือชื่อนิสิต........อัจฉรา มหาวีรวัฒน์........
สาขาวิชา..วิทยาการคอมพิวเตอร์.............	ลายมือชื่ออาจารย์ที่ปรึกษา....[illegible]
ปีการศึกษา 2549..............................	ลายมือชื่ออาจารย์ที่ปรึกษาร่วม..........[illegible]........

4373866623 : MAJOR COMPUTER SCIENCE
KEY WORD: SOFTWARE FAULT/ PREDICTIVE MODEL/ FAULT METRICS/ FAULT PREDICTION/ FAULT IDENTIFICATION/ NEURAL NETWORKS/ FUZZY LOGIC/ RADIAL-BASIS FUNCTION/ FUZZY SUBTRACTIVE CLUSTERING/ DYNAMIC FAULT/ PROGRAM STRUCTURE GRAPH/ FAULT LOCATION/ SENSITIVITY ANALYSIS

ATCHARA MAHAWEERAWAT : SOFTWARE FAULT PREDICTION BY FUZZY LOGIC AND NEURAL NETWORK TECHNIQUES THESIS ADVISOR : ASSOC. PROF. PERAPHON SOPHATSATHIT, Ph.D., THESIS CO-ADVISOR : PROF. CHIDCHANOK LURSINSAP, Ph.D., 180 pp. ISBN 974-14-3911-3.

In the world of software development, organizations must optimize the use of their limited resources to deliver quality products on time and within budget. This requires efficient and effective discovery, removal, and prevention of faults introduced during the development process or residual faults from maintenance stage. To reveal software fault, testing is generally employed by procedurally running the system with adequate test cases. Such an undertaking usually incurs high costs and considerable efforts. This dissertation proposes an approach for software fault prediction and fault location without actually running the software. The process of software fault prediction consists of four stages, namely, fault-prone prediction, fault type prediction, dynamic fault prediction, and fault locating. Fault predictive models are constructed based on software metrics with the help of fuzzy logic and neural network techniques for each stage. Once identified, all potential faults are pinpointed to locate their whereabouts. The results are further analyzed to obtain principal metrics that are conducive to fault prediction with the help of sensitivity analysis process. Hence, the proposed approach will furnish a basis for machine learning building blocks that could be realized in software quality assurance, whereby replacing time-consuming and error-prone inspection process to attain more reliable software products.

Department……Mathematics……………Student's signature…อัจฉรา มหาวีรวัฒน์

Field of study….Computer Science……..Advisor's signature…[illegible]

Academic year 2006……………………Co-advisor's signature…[illegible]

Acknowledgements

I wish to thank the teachers who helped shape my scientific purview and educational foundation. Two of them stand out. My thesis advisor, Assoc. Prof. Dr. Peraphon Sophatsathit, who strengthened my knowledge of software engineering and research methodology. He suggested a lot of valuable ideas and encouraged me to complete the dissertation. I appreciate my thesis co-advisor, Prof. Dr. Chidchanok Lursinsap for introducing me to the world of neural networks and fuzzy logic. He inspired me to incorporate knowledge of neural networks, fuzzy logic, and software engineering to complimenting this dissertation.

I would like to thank Assoc. Prof. Dr. Petr Musilek for providing some computing facilities and the pleasant atmosphere he created when I visited FACIA (the Facility for Advanced Computational Intelligence and Applications), University of Alberta, Canada.

I am indebted to Assoc. Prof. Dr. Filippo Lanubile for his kindness of giving me the data for my first experiment.

Also, it is my pleasure to acknowledge the necessary financial support from The Office of Higher Education Commission, Ministry of Education, Thailand.

I would like to recognize the efforts of AVIC members. They provided additional comments that helped improve this dissertation and contribute to the congenial environment of the Advanced Virtual and Intelligent Computing (AVIC) research center.

Finally, a special thank to my parents, brother and sisters, and Mr. Suphakant Phimoltares for their love and support.

Contents

Chapter page

List of Tables

Table page

List of Figures

Figure page

Figure page

List of Abbreviations

RBFN	Radial-Basis Function Network
NN	Neural Network
MLP	Multilayer Perceptron
BP	Back-Propagation
OO	Object-Oriented
SDA	State Definition Anomaly (possible post-condition violation)
SDIH	State Definition Inconsistency (due to state variable hiding)
SDI	State Definition Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
SVA	State Visibility Anomaly
PSG	Program Structure Graph
AAG	Attribute Access Graph
MIG	Method Invocation Graph
IFG	Inheritance Flow Graph
ASG	Association Graph
CFG	Control Flow Graph
SA	Sensitivity Analysis

List of Symbols

⯃	Class
⬭	Attribute
▢	Method
⬯	Referred inherited attribute
⬚	Referred inherited method
$<OP_{si},OP_{bi}>$	Vertex tag
◌	Global function
○	Source code section
—◇	Private membership
┄◇	Protected membership
– –◇	Public membership
—▷	Class inheritance
⋯➤	Member inheritance
- - -➤	Method invocation
- - -➤	Attribute access
⟶	Private inheritance
⋯>	Protected inheritance
-·-·->	Public inheritance
⋯⊳	Friend Association
—▶	Control flow
– –▷▷	Involving
⇨	Inheritance path
▬ ➜	Association path

CHAPTER I

INTRODUCTION

1.1 Background and Motivation

The proliferation of software application has involved in various facets of daily living to the extent that many undertakings cannot do without it. As such, reliability is utmost important to the operations of software application. Software reliability is the probability of failure free operations of a computer program executing in a specified environment for a specified time [1]. Software reliability is considered a software quality factor that aids in predicting software quality using standard predictive models. There are many approaches for predicting software quality, most of which yield some forms of quality indicators. One popular indicator is known as software fault. Software fault prediction utilizes historical and development data to arrive at a conclusive decision whether the software in question is at fault.

Software faults have plagued the quality and reliability of software systems since their inception. Despite a consensus in the imperfect software process, software faults can be reduced with precautionary efforts. Numerous "fault-free" endeavors ranging from software inspection, cleanroom development to formal methods have been instituted to improve software reliability. It is, therefore, essential that these faults be detected, located, and eventually removed from the software, whereby enhancing software reliability to an acceptable level. Unfortunately, the above reliable and rigorous approaches are not practical in large software development system due to human error. As a consequence, some forms of automated fault prediction mechanisms must be devised to accommodate such short falls.

Recent research and development in fuzzy logic and neural networks have improved the accuracy and efficiency of fault prediction considerably. Fuzzy Logic provides a means for determining whether the indecisive circumstance is right or wrong. Neural Networks, on the other hand, offers an automated means for carrying out desired operation without human intervention. As a consequence, the two techniques are utilized in this research work to arrive at the proposed software fault detection method.

1.2 Dissertation Overview

This dissertation proposed an approach for software fault prediction as done in conventional manual inspection. However, the principal difference is that the proposed approach employs machine learning modelling to construct a prototype for automatic software fault prediction. This permits large scale inspection where voluminous code (that would otherwise be impossible to carry out manually) is inspected, whereby potential faults can be collected, compiled, and analyzed for the causes. The prediction process starts from measuring software components with software metrics. Then fault prediction is performed applying predictive model based on software metric values as demonstrated in Figure 1.1. The proposed process can be divided into four stages, namely, (1) fault-prone prediction, (2) fault type prediction, (3) dynamic fault prediction, and (4) fault locating as demonstrated in Figure 1.2. The fault-prone prediction is the first stage where software components (hereafter referred as classes) are explored whether they are faulty or fault-free classes [2]. Then the faulty classes are further analyzed to identify fault type, namely, SDA, SDI, SVA, SDIH, and IISD [3] in the stage of fault type prediction [4, 5]. Since both earlier stages predict software faults based on

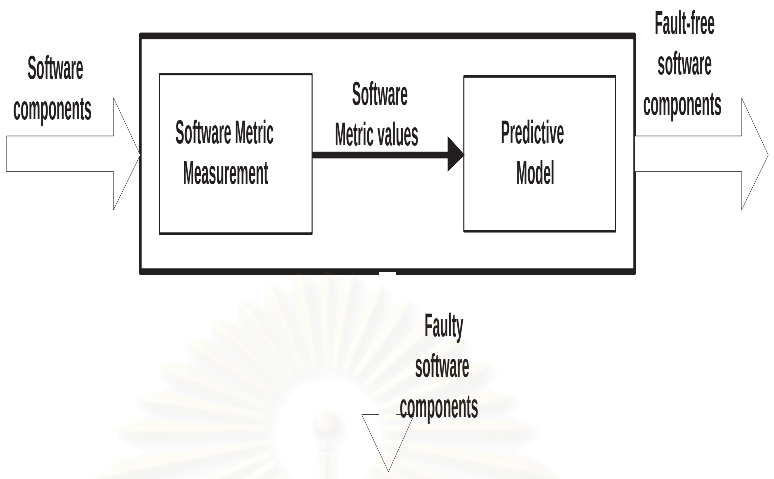

Software components
Software Metric Measurement
Software Metric values
Predictive Model
Fault-free software components
Faulty software components

Figure 1.1: Diagram of software fault prediction approach

software metric gathered from source code, they are considered as a static fault analysis.

However, there exists faults which occur at runtime and may not be detected at static fault analysis stage. Consequently, the predicted fault-free classes from the first stage are passed to dynamic fault prediction stage. At this stage, software classes are represented by a set of graphs. Some metrics are extracted from those graphs and applied to faultiness prediction.

In addition to faultiness, the location of fault is also investigated by finding faulty methods and attributes based on a set of method and attribute metrics. Details of software prediction process are described in the chapters that follow.

1.3 Objectives

The objectives of this dissertation are

1. To develop an approach to predict fault in production software using fuzzy logic and neural network techniques

2. To propose a guideline for locating software fault in software development process

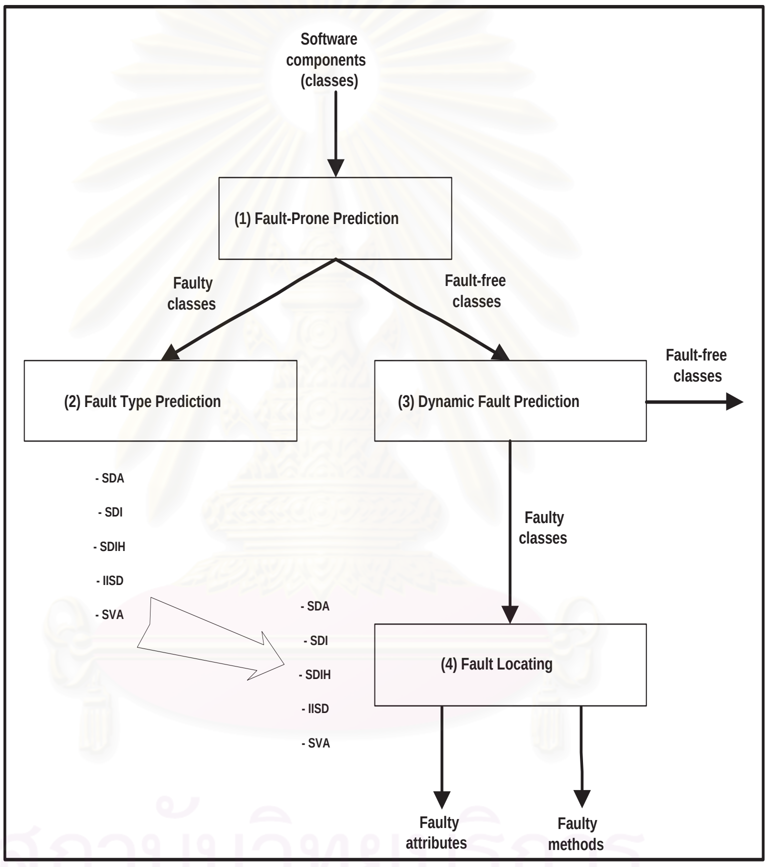


Figure 1.2: Stages of software fault prediction process

1.4 Scope of Work

This dissertation focuses on C++ programs developed based on both structured and object-oriented programming approaches as case studies. The case study programs were written in simple style. Some software faults due to inheritance are also considered. The programming language and tools employed in this study are Matlab, free/commercial software metrics and graph tools.

1.5 Contributions

The proposed software fault prediction approach contributes to software development system as follows:

(1) Furnish faultiness predictive model for detecting fault-prone software;

(2) Provide fault type identification model for identifying fault type;

(3) Represent software as a set of graphs and extract metrics from those graphs;

(4) Locate the cause of fault by finding faulty methods and attributes; and

(5) Gauge how important each metric is conducive toward the cause of faultiness by means of a proposed algorithm and sensitivity analysis.

1.6 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents existing related works. Chapter 3 describes theoretical background. The first phase of software fault prediction. Fault-prone prediction is detailed in Chapter 4. Fault type identification is subsequently explained in Chapter 5. Chapter 6 presents dynamic fault prediction. Some conclusive results and future work are summarized in Chapter 7.

CHAPTER II

RELATED WORKS

There have been a number of researches employing neural networks and fuzzy logic to predict software error or software fault. Some empirical comparative studies in neural network versus statistical methods are presented in [6–10]. Evidence of research data are collected from many sources such as parameter values released from executed system [11], software metrics [10, 12–15], and software change history [16].

Fuzzy logic and neural network techniques are primarily used to analyze data and infer the relationship among them. Pedrycz, Succi, Reformat, Musilek and Bai [17] used Self Organizing Maps (SOM) to divide software into clusters to explicitly capture relationships between the software measures and quantify these dependencies for larger and less homogeneous clusters of software modules. Khoshgoftaar used back-propagation learning algorithm and Principal Component Analysis (PCA) technique [18, 19] to predict software faults. Yuan [20] applied fuzzy subtractive clustering technique with fuzzy inferences and some statistical techniques to predict software quality.

Toshihiro Kamiya et al. [21] proposed a method to estimate the fault-proneness of software classes in the early phase using several complexity metrics and multivariate logistic regression analysis. They introduced four checkpoints into the analysis/design/implementation phase, and estimated the fault-prone classes using applicable metrics at each checkpoint.

In Briand [22], a fault-proneness prediction model was built based on a set of object-oriented measures using data collected from a mid-size Java system with the help of logistic regression analysis.

P. Kokol et al. [23] introduced some methods for reliability prediction based on a large database of modules in C language using a set of selected software metrics. The results and methods were compared. They found that statistical and mathematical methods accurately predicted the reliability of software modules, whilst black box approach could not explain the reasons behind the prediction.

In Aljahdali [9], neural networks were proposed as an alternative technique to build software reliability growth models. A comparison between regression parametric model and neural network model was carried out and concluded that neural networks were able to provide models with small Sum-Squared Error (SSE) better than the regression model in all considered cases.

Mie Mie Thet Thwin and Tong-Seng Quah [24] presented the application of neural networks for predicting the number of faults in three industrial real-time systems based on object-oriented design metrics. Ward Network which is a back-propagation 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. [25] 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 use in predicting if future releases of a commercial Java classes would be faulty.

Lionel C. Briand et al. [26] 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. [27] performed an empirical study with the data obtained from a commercial Java application using logistic regression technique. They found that Depth of Inheritance Tree (DIT) was a good measure of familiarity and had a quadratic relationship with fault-proneness.

Yida Mao, H.A. Sahroui, and Hakim Lounis [28] 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 [29] 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 [30] analyzed more than 200 different object-oriented metrics extracted from the literature with the aim of identifying suitable models for detecting fault-prone classes. The work was focused on identifying models that could detect as many faulty classes as possible based on a manageable small set of metrics. To reach their goal, Principal Component Analysis was applied to find the subset of metrics, whereby multivariate logistic regression analysis was subsequently applied to construct the models. Besides the prediction of fault-proneness in object-oriented software, fault type is also detected in [31]. Roger T. Alexander et al. [31] defined a set of experiments, encompassing relative effectiveness of several coupling-based O-O testing criteria and branch coverage. All O-O testing criteria were more effective at detecting faults due to the use of inheritance than branch coverage.

CHAPTER III

THEORETICAL BACKGROUND

3.1 Software Reliability and Prediction Models

The proliferation of computer technology has brought about increasingly complicated software demand. As complexity grows, so does quality needs. Unfortunately, complexity works against quality, thus resulting in low reliability. New software development paradigms are employed to cope with such stringent requirements, for example, zero-defect software quality assurance, software fault detection, and software reliability measurement, etc.

Software reliability involves various precautions to guard against faulty operations, ranging from testing, production, and maintenance. As it is generally known that exhaustive test is in no way practical, failure is thus inevitable. In principle, software failure is defined as the departure of the external results of program operation from equipments. Such discrepancies are referred to as faults.

A fault is the defect in the program that causes failure [1]. Faults can be classified based on [32] as follows:

- *Locality* which includes atomic component faults, composite component faults, system level faults (i.e., operator faults, replication faults), and external faults (i.e., environment faults, user faults);

- *Effect* which includes value faults (a result from a computation that does not meet the system specification), timing faults (a processor or service which is not delivered or completed within the specified time interval);

- *Cause* which includes immediate cause (such as resource depletion faults that involve a section of the system being unable to obtain the resources required to perform its task, logic faults that result from the system not behaving according to specification, physical faults that are caused by hardware breaks or mutation in executable software), ultimate cause (specification faults, configuration faults);
- *Duration* which includes persistent faults (that remain active for a significant period of time), transient faults (that remain active for a short period of time); and
- *Effect* on system state which is characterized by faults describing the system states.

A study [1] shows that there are two principal factors that affect the behavior of failure. The first factor is the number of faults in the software being executed, and the second factor concerns with the execution environment or operational profile of execution. The fact that software development process is still a human-oriented activity makes it impossible to produce fault-free software products. As such, faults are usually introduced when the code is being developed by programmers, during original design, adding new features or design changes, or fault repair. Various efforts have been taken to remove new faults being introduced during maintenance, namely, regression test, cleanroom technique, etc. The removal process often take places when the first fault is detected. This process, from a practical standpoint, is dependent on the efficiency which faults are found and removed.

To assess how effective the above approach is to software reliability, it is necessary that some quantification analyses be instituted. Such an assessment measure is known as software reliability measure. One simple approach to measure software reliability is to count the number of failures occurred within a specified time. This number is used to predict the cause of failures through software fault prediction model. Construction of the prediction model is based on historical information which is the data collected from past software products properties. The model so constructed should encompass some essential characteristics as follows [1]:

- gives good predictions of future failure behavior,
- computes useful quantities,
- is simple,
- is widely applicable, and
- is based on sound assumptions.

To model software reliability, we considered the process involving the above principal factors from three viewpoints, namely, *fault introduction, fault removal*, and *the environment*. Fault introduction depends primarily on the characteristics of the developed code (code created or modified for the application) and characteristics of the development process [1]. The most significant code characteristic is size, whereas the development process characteristics encompass software engineering technologies, tools, and levels of experience of personnel. Such a software reliability model specifies a general failure process dependency based on the aforementioned factors. This dependency can be defined by establishing the parameters of the model through estimation or prediction. The former rests on statistical techniques being applied to failure data taken from the programs, whereas the latter determines the properties of software product quality and the development process using fuzzy logic or neural network techniques.

The prediction approach often incorporates future failure behavior to anticipate unforeseen faults. Unfortunately, the primary assumption requires that the model's parameters not change for the period of prediction. If any change is made to the parameters, the result of the prediction will not be accurate. Thus, many software reliability models are based on stable program executing in the underlying constant environment.

Fault removal uses verification and testing techniques to locate faults, thus enabling the necessary changes to be made to the system [32]. Fault removal cannot occur without the ability to detect faults in the first place. Thus fault removal is dependent on the detection efficiency which faults are found and removed [1].

The environment is described by the operational profile which is defined as a set of run types that a program can execute, along with the probability the run type will occur [1]. The term 'run' is generally associated with some functions that the program performs.

The benefits from software reliability can be further applied in many software activities such as [1]

- evaluate software engineering technology quantitatively,
- evaluate development status during the test phases of a project,
- monitor the operational performance of software,
- control new features added and design changes made to software, and
- view insight the software product and the software development process.

In this dissertation, the count of software fault and selected software fault prediction parameters (or metrics) were employed as the bases for the proposed reliability model. Software reliability analyses can then be carried out by means of the proposed model operating on sample code.

3.2 Software Metrics

Software is a complex artifact which requires elaborate methods of measurement to gauge its complexity. The measured results are often applied to further software development and improvement quantitatively and qualitatively.

Basically, there are three classes of metrics used in conventional software measurement, namely, process metrics, product metrics, and resource metrics [33]. The process metrics focus on any software related activities which may be part of the software development cycle, maintenance, and retirement. Further applications of the process metrics in recent team development such as the maturing process of software development organizations, personal software development process, and small team development such as eXtreme Programming. Process metrics can be further classified into subclasses as follows [10]:

- Maturity Metrics
- Management Metrics
- Life Cycle Metrics

The product metrics generally describes the characteristics of software products in terms of size, complexity, design features, performance, and quality level. We shall expand some of these metrics classifications below [10].

- Size Metrics
- Architecture Metrics
- Structure Metrics
- Quality Metrics

- Complexity Metrics

The final metric, the resource metrics, encompasses anything that involves the activities in the software production process including personnel, materials, and methods. They are [10]:

- Personnel Metrics
- Software Metrics
- Hardware Metrics

Since object-oriented aspect has been applied to software development for decades, metrics for object-oriented software have been proposed. Object-oriented metrics are developed to realize the structure and characteristics of object-oriented programs by many researchers such as Chidamber and Kemerer [34]. Some of their metrics are described below:

Inheritance related measures

- **Depth of Inheritance Tree (DIT)** of a class is the length of the longest path from the class to the root in the inheritance hierarchy. This determines the complexity of a class based on its ancestors, since a class with many ancestors is likely to inherit much of their complexity. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit. This makes it difficult to predict the class behavior.
- **Number of Children (NOC)** measures the number of immediate descendants of a particular class. This measures an amount of potential reuse of the class. The more reuse a class might have, the more complex it may be, and the more classes are directly affected by changes in its implementation.

Coupling measures. Coupling metrics measure the degree of inter-dependence among the components of a software system. High coupling makes a system more complex and harder to understand, change, or correct.

- **Coupling Between Objects (CBO)** is defined as the number of other classes to which it is coupled.
- **Response For a Class (RFC)** is the number of methods that can potentially be executed in response to a message received by an object of that class. The response set of a class consists of the set of M methods of the class, and the set of methods directly or indirectly invoked by methods in M.

Complexity measures

- **Weighted Methods per Class (WMC)** is defined as being the number of all member functions and operators defined in each class.

3.3 Object-Oriented Concepts

Objects are the physical and conceptual things found in the universe. Hardware, software, documents, human beings, and even concepts are all examples of objects. Object-oriented software is all about objects. An object is a "black box" which receives and sends messages. A black box actually contains code (sequences of computer instructions) and data (information on which the instructions operates). Class is an abstract representation of a particular type of object and often described as a plan or blueprint for an object, as opposed to the actual object itself. There are many object-oriented concepts introduced for object-oriented software development. Application of object-oriented concepts can lead to fault occurrence in software [35]. As a consequence, this dissertation focuses on the object-oriented concepts relevant to software fault as follows:

3.3.1 Inheritance

Inheritance is the property where one class incorporates another class methods and/or attributes. The original class is called the superclass of the inherited or extending class, and the extending class is called the subclass of the superclass. Since a subclass contains all data and methods of the superclass plus additional resources it creates, it is more specific; conversely, since the superclass lacks some of the resources defined in the subclass, it is more general or abstract, albeit less detailed than its subclasses. Inheritance comes in two forms, namely, single inheritance and multiple inheritance [36]. In single inheritance, each subclass inherits from no more than one superclass. On the other hand, multiple inheritance allows the subclass to inherit from more than one immediate superclass or parent. The rules for defining multiple inheritance must handle any conflicts which may arise. For example, naming conflicts can arise where the same name may be used to represent different attributes or methods in two different parents and both these attributes or methods are inherited by the same child class. Many object-oriented programming languages can handle this naming conflict, except when some kind of naming conflicts occur in a subclass defining an attribute or method with the same name as inherited attribute or method from its ancestor classes [3, 35].

3.3.2 Association

An association relationship between two classes means that one class members can be used in the other's members. The "uses" class sends a message or its instance to an instance or a member function of the "used" class [37, 38]. There are four types of association relationships [39]:

- Friend member function association
- Friend class association

- Friend operation association is the association between classes through a global function.
- Ordinary association is the association established through parameter passing of an instance of one class to a member function of another class.

More details of two above concepts, namely, inheritance and association, are described in Chapter 6.

3.4 Fuzzy Subtractive Clustering

Fuzzy logic is an extension of standard Boolean logic [40]. The fundamental concepts is a fuzzy set in which each element in the set is characterized by its grade of membership of the set [41]. A membership function maps an element of the set in a given domain to an appropriate membership grade value, ranging between zero and one. Typical membership functions are the Gaussian distribution function, the sigmoid curve quadratic, cubic polynomial curve, and etc.

A cluster is a group of entities with similar properties [42]. Entities from different clusters cannot be similar. Thus, a set of points can be grouped into one or more clusters, depending on the properties of those points. Usually, the obvious property is the distance among the points. Unfortunately, separating these points into clusters is, in many cases, not straightforward since there are points that lie between different clusters of points which could belong to either adjacent clusters. This is where fuzzy clustering comes into play. To determine if a point belongs to a given cluster, every point in the cluster must lie within a given proximity. Thus, the distance between the point in question and its neighboring points must be smaller than the distance to the rest of the points. In addition, points in different clusters are assigned different grades of membership based on the selected membership function. Such an assignment calls for an algorithm that must be able to compute the membership grade value in a finite number of iterations. This is a tall order that ordinary distance algorithmic approaches cannot accomplish

There are many fuzzy clustering techniques to group data points into clusters. One popular technique is Fuzzy C-Means clustering which is a simple and straightforward approach. This technique, however, requires predefined number of clusters where every data point membership depends on a membership grade. The subtractive clustering, on the other hand, does not require a predefined number of clusters. The subtractive clustering is a fast, one-pass algorithm for estimating the number of clusters and cluster centers in a set of data [42]. Each data point is considered a potential cluster center, which is calculated from the density of the surrounding data points [20]. Given a group of data points $\{x_1, x_2, \ldots, x_n\}$, all data points are normalized with respect to each variable (vector) associated with x_i, the initial potential value of the data point x_i is defined as [43]

$$P_i = \sum_{j=1}^{n} e^{-\alpha \|x_i - x_j\|^2} \tag{3.1}$$

where	α	$= \frac{4}{r_a^2}$
	$\|\cdot\|$	is the Euclidean distance
	r_a	is a positive constant

The constant r_a is effectively a normalized radius defining a neighborhood, data points outside this radius have little influence on the potential. The data point with the highest potential value is selected to be the first cluster center.

Let x_1^* be the first cluster location with the potential value P_1^*. The potential value of each data point x_i is revised according to Equation (3.1).

$$P_i = P_i - P_1^* e^{-\beta \left\| x_i - x_1^* \right\|^2} \tag{3.2}$$

where $\beta = \frac{4}{r_b^2}$, r_b is a positive constant.

The constant r_b is normally larger than r_a (suggested to be 1.25 [44]) to avoid obtaining closely spaced cluster centers. The potential value of each data point near the first cluster center thereby is reduced by Equation (3.2) so that they will unlikely be chosen as the next cluster center [43].

The data point with the highest remaining potential is obtained and set as the next cluster center. The potential of the remaining data points is then recalculated according to their distance to the new cluster center. The general potential revision formula [43] can be expressed according to Equation (3.3).

$$P_i = P_i - P_k^* e^{-\beta \left\| x_i - x_k^* \right\|^2} \tag{3.3}$$

where x_k^* is the location of the k^{th} cluster center, P_k^* is its potential value.

The procedure of finding a new cluster center is repeated until a sufficient number of cluster centers are generated. The stopping criterion is that the remaining potential of all data points are lower than some fractions of the first cluster center potential P_1^* which is usually set to $P_k^* < 0.15 P_1^*$ as suggested in [43, 44].

3.5 Supervised-Learning Neural Network

3.5.1 Multilayer Perceptron Network

Multilayer perceptrons are multilayer feedforward networks. The network consists of a set of sensory units that constitute an input layer, one or more hidden layers of computation nodes, and an output layer of computation nodes as demonstrated in Figure 3.1.

The input signal propagates through the network in a forward direction, on a layer-by-layer basis.

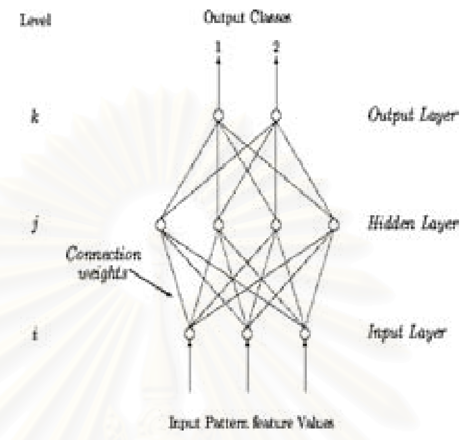


Figure 3.1: Architecture of multi-layer perceptron

There are two kinds of signals in this network.

1. *Function signals.* A function signal is an input signal that comes in at the input end of the network propagating forward (neuron by neuron) through the network and emerges at the output end of the network as an output signal. When a function signal passes each node, the entering signal is transformed according to the associated weights of the input arc to that node. The transformed signal is further processed by an activation function of that node, resulting in an output function signal emerging out of that node. At the output layer, the output signal function becomes the output of the network.

2. *Error signals.* An error signal originates at an output neuron of the network, and propagates backward (layer by layer) through the network. The hidden or output neuron of a multilayer perceptron performs two computations

(a) A continuous nonlinear function of the input signal and synaptic weights associated with that neuron.

(b) An estimate of gradient vector which is needed for the backward pass through the network.

Multilayer perceptron is the most widely used neural network and has also been applied to solve some difficult problems by algorithmically training them in a supervised manner. There are many learning algorithms that are developed for multilayer perceptron networks, notably one of those is back-propagation learning algorithm.

Back-propagation Learning Algorithm

Error back-propagation learning consists of two passes through the different layers of network, namely, forward pass and backward pass as described below.

1. Forward pass. An activity pattern is applied to the input node of the network. Its effect propagates through the network layer by layer. Upon the existing network, a set of outputs is produced as the actual response of the network.

2. Backward pass. An error signal is produced by subtract the actual response of the network from the desired response and is then propagated backward through the network. The synaptic weights which are all fixed in the forward pass, are adjusted to minimize the difference between the actual response and the desired response of the network.

For any given successive layers, the input to each node is the sum of the scalar products of the incoming vector components with their respective weights. Thus the input to a node j (Figure 3.1) is given by

$$input_j = \sum_i w_{j,i} \cdot out_i \tag{3.4}$$

where w_{ij} is the weight connecting node i to node j and out_i is the output from node i. The output of a node j is determined by

$$out_j = f(input_j) \tag{3.5}$$

and this output is sent to all nodes in the following layer. This computation is continued through all the layers of the network until the output layer is reached and the output vector is produced.

The function f is an activation function of each node. For this work, sigmoid activation function is used which can be expressed as

$$f(x) = \frac{1}{1 + exp(-x)} \tag{3.6}$$

where $x = input_j$. So the node acts like a thresholding device according to the sigmoidal curve illustrated in Figure 3.2.

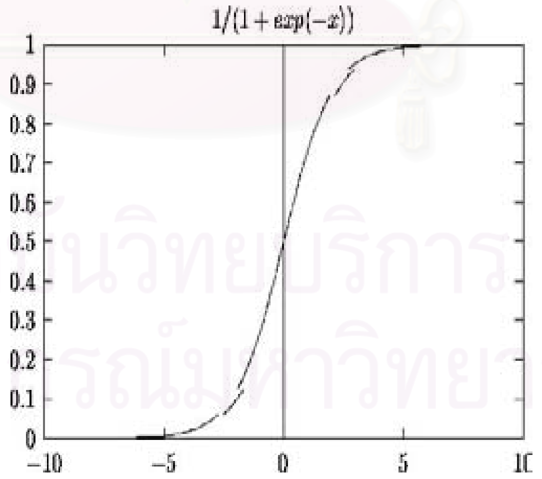


Figure 3.2: The sigmoid activation function.

In learning phase, a set of input patterns (training set) initialized with small random values are presented at the input layer, together with their corresponding desired output patterns. Each input pattern is then passing through the input layer, where the weights attached to the connections are adjusted accordingly. This reduces the difference between the network's output and the desired output for that input pattern. The weights between the output layer and the proceeding layer (hidden layer) are adjusted by the generalized delta rule (to be described later) based on the difference error term or δ term in the output layer as follows:

$$w_{kj}(n+1) = w_{kj}(n) + \eta(\delta_k out_k) \tag{3.7}$$

where $w_{kj}(n+1)$ and $w_{kj}(n)$ are the weights connecting nodes k and j at iteration $(n+1)$ and n, respectively, η is a learning rate parameter. Subsequently, the δ term for hidden layer nodes are computed and the weights connecting the hidden layer with the previous layer (another hidden layer or input layer) are updated. This calculation is repeated until all weights in the last layer have been adjusted.

The previous equation can be referred as the rate of change of error with respect to the input to node k, and can be written as

$$\delta_k = (d_k - out_k)f'(input_k) \tag{3.8}$$

for nodes in the output layer, and

$$\delta_j = f'(input_k)\sum_k \delta_k w_{kj} \tag{3.9}$$

for nodes in the hidden layers, where d_k is the desired output for node k.

The concepts behind back-propagation algorithm is a gradient descent optimization procedure which minimizes the mean squared error between network's output and the desired output for all input patterns P, that is,

$$E = \frac{1}{2P} \sum_{p} \sum_{k} (d_k - Out_k)^2 \tag{3.10}$$

The training set is presented iteratively to the network as the weights are updated until their value are stable and the following criteria are reached:

- a user defined error-tolerance is achieved, or
- a maximum number of iterations is completed.

Summary of training procedure for multilayer perceptron feed-forward neural networks is shown in Figure 3.3.

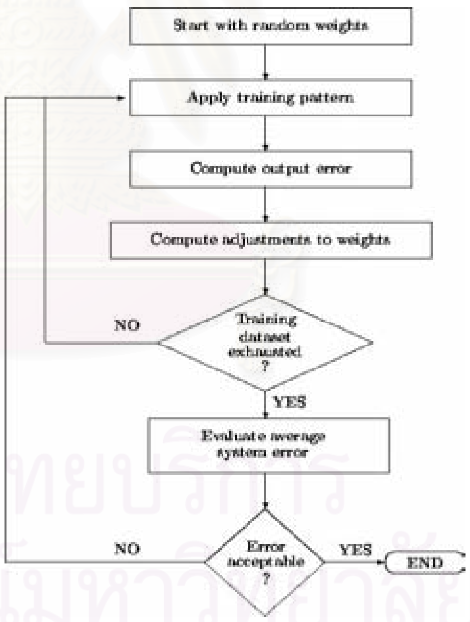

Start with random weights
Apply training pattern
Compute output error
Compute adjustments to weights
NO
Training dataset exhausted ?
YES
Evaluate average system error
NO
Error acceptable ?
YES
END

Figure 3.3: Training procedure for multilayer perceptron network.

Experimental data are prepared in two sets, namely, training set and test set. The training set is used to establish various network parameters, while the test set is used to adjust those parameters, as well as to assess the performance of the network during and after training. Once trained, the configuration so established is saved for use in classification phase.

The Generalized Delta Rule

Opertion of the network requires that the weights be adjusted iteratively in order to minimize the mean-squared error. The procedure corresponding to such action is a gradient descent optimization in weight space. Consider the symbols in Figure 3.1, when a pattern p is presented to the network, the weight changes are given by

$$\Delta w_{kj} = -\eta \frac{\partial E_p}{\partial w_{kj}} \tag{3.11}$$

where E_p is the squared error for input pattern p and is given by

$$E_p = \frac{1}{2} \sum_k (d_k - out_k)^2 \tag{3.12}$$

and $E = \sum_p E_p$ is the mean squared error as given in Equation (3.10). Applying chain rule to right hand side terms of Equation (3.11) becomes

$$-\frac{\partial E_p}{\partial w_{kj}} = -\frac{\partial E_p}{\partial input_k} \frac{\partial input_k}{\partial w_{kj}} \tag{3.13}$$

This means that the weight changes can be expressed as the product of two terms. From Equation (3.4), the rate of change of error with respect to input to node k and the change of input to node k with respect to a change in weight between nodes k and j.

$$\frac{\partial input_k}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \sum w_{kj} out_j = out_j \tag{3.14}$$

Let $\delta_k = -\frac{\partial E_p}{\partial input_k}$, then Equation (3.11) becomes

$$\Delta w_{kj} = -\eta \delta_k out_j \tag{3.15}$$

The term δ_k can be expressed as the following equation by using chain rule,

$$\delta_k = -\frac{\partial E_p}{\partial input_k} = -\frac{\partial E_p}{\partial out_j}\frac{\partial out_k}{\partial input_k} \tag{3.16}$$

From Equation (3.5)

$$\frac{\partial out_k}{\partial input_k} = f'(input_k) \tag{3.17}$$

Substituting Equation (3.12) and (3.17) in Equation (3.16) become

$$\delta_k = (d_k - out_k) f'(input_k) \tag{3.18}$$

for any output layer node k and a given pattern p. Using chain rule, a node in any hidden layer is expressed as

$$\delta_j = f'(input_j) \sum_k \delta_k w_{kj} \tag{3.19}$$

Applying Cross Validation to Stopping Criteria

In a neural network training phase, the important goal is to obtain optimal generalization performance. Generalization performance means small errors on samples will not be seen during training. Unfortunately, standard neural network architectures such as the multilayer perceptron are prone to overfitting. This phenomenon takes place during training as the network seems to perform better (the error on the training set decreases), at some point it reverts to get worse again (the error on unseen examples increases).

There are two ways to overcome overfitting, the first way is reducing the number of dimensions of the parameter space or the effective size of each dimension. The other way is early stopping. Early stopping can be used either interactively based on human judgement, or automatic stopping criterion.

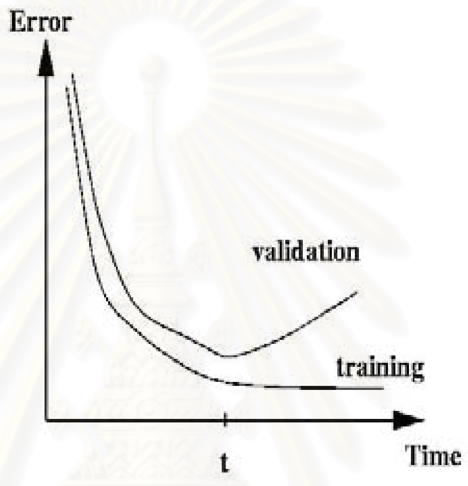


Figure 3.4: Idealized training and generalization error curves

Figure 3.4 shows the evolution overtime of the per-example error on training set and test set. From the behavior shown in the figure, the following steps describe how to apply cross validation to early stopping:

- split the training data into a training set and a cross validation set,
- train only on the training set and evaluate the per-example error on the validation set once in a while,
- stop training as soon as the error on the cross validation set is higher than the last time the error was cross validated, and
- use the weights of the network obtained from the previous training run.

This method uses the cross validation set to simulate the behavior on the test set, assuming that the errors of both sets are similar.

There are a number of plausible proposed stopping criteria proposed that can be classified into three classes by Lutz Prechelt [45]. From this paper, some definitions are defined. Let

- E be the objective function (error function) of the training algorithm,
- $E_{tr}(t)$ be the average error per example over the training set, measured after epoch t,
- $E_{va}(t)$ be the corresponding error on the validation set and be used by the stopping criterion,
- $E_{te}(t)$ be the corresponding error on the test set,
- $E_{opt}(t)$ be the lowest validation set error obtained in epochs up to t: $E_{opt}(t) = \min_{t' \leq t} E_{va}(t')$

The generalization loss at epoch t is the relative percentage increase of validation error over the minimum-so-far (in percent) [45]:

$$GL(t) = 100 \times \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right) \tag{3.20}$$

A high generalization loss directly indicates overfitting, which is the reason to stop training. Consequently, the first class of stopping criteria is defined as (GL_α): stop as soon as the generalization loss exceeds a given threshold; alternatively, GL_α: stop after first epoch t with GL(t) $> \alpha$.

When training error rate decreases quickly, no overfitting takes place. As the rate levels out, overfitting shapes up. A training strip of length k is defined as a sequence of

k epochs numbered $n+1, ..., n+k$, where n is divisible by k. The training progress (in per thousand) measured after such a training strip is

$$P_k = 1000 \times \left(\frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \times \min_{t'=t-k+1}^{t} E_{tr(t')}} - 1 \right) \tag{3.21}$$

This means that how much the average training error during the strip was larger than the minimum training error during the strip

The second class of stopping criteria is defined as the quotient of generalization loss and progresses as follows [45]:

PQ_α: stop after first end-of-strip epoch t with $\frac{GL(t)}{P_k(t)} > \alpha$

The cross validation error is measured only at the end of each strip.

The third class of stopping criteria is defined depending on the sign of changes in the generalization. The rationale is to stop when generalization errors increases in s successive steps.

UP_s: stop after epoch t if UP_{s-1} stops after epoch $t-k$ and $E_{va}(t) > E_{va}(t-k)$

UP_1: stop after first end-of-strip epoch t with $E_{va}(t) > E_{va}(t-k)$

This definition means that when the validation error has increased not only once, but during s consecutive strips and assume that such increases indicate the beginning of final overfitting, independent of how large the increases actually are.

From all classes (GL, PQ, and UP), the class PQ is employed as the stopping criterion of the training with back-propagation learning approach.

3.5.2 Radial-Basis Function Network

The Interpolation Problem

The radial-basis function networks (RBFN) mention that the problem of curve-fitting is approximation in high dimensional spaces. In this case, the learning process is equivalent to finding an interpolating surface in the multidimensional space that provides the best fit to the training data, measured by pre-selected statistical criteria.

The curve-fitting or interpolation problem can be stated as follows:

Given a set of N different points $\{x_i \,\epsilon\, \mathcal{R}^{m_0} \,|i = 1, 2, ..., N\}$ and a corresponding set of N real numbers $\{d_i \,\epsilon\, \mathcal{R}^1 \,|i = 1, 2, ..., N\}$, find a function $F : \mathcal{R}^N \rightarrow \mathcal{R}^1$ that satisfies the interpolation condition:

$$\mathcal{F}(x_i) \;=\; d_i \quad i = 1, 2, \ldots, N \tag{3.22}$$

Radial-Basis Functions

The radial-basis functions (RBF) technique suggests that the interpolation function $\mathcal{F}$ should be constructed in the following form

$$\mathcal{F}(x) \;=\; \sum_{i=1}^{N} w_i \varphi(\|x - x_i\|) \tag{3.23}$$

where $\{\varphi(\|x - x_i\|) \,|i = 1, 2, \ldots, N\}$ is a set of N arbitrary (generally nonlinear) functions; radial-basis functions; and $\|\cdot\|$ is the Euclidean norm. The known data points $\{x_i \,\epsilon\, \mathcal{R}^{m_0} \,|i = 1, 2, ..., N\}$ are defined to be the centers of the radial-basis functions.

Inserting the interpolation conditions of Equation (3.22) in Equation (3.23), a set of simultaneous linear equations for the coefficients (weights) of the unknown w_i are expanded as follows:

$$\begin{bmatrix} \varphi_{11} & \varphi_{12} & \cdots & \varphi_{1N} \\ \varphi_{21} & \varphi_{22} & \cdots & \varphi_{2N} \\ \vdots & \vdots & & \vdots \\ \varphi_{N1} & \varphi_{N2} & \cdots & \varphi_{NN} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} \tag{3.24}$$

where

$$\varphi_{ji} = \varphi(\|x_j - x_i\|) \quad (j,i) = 1,2,\ldots,N \tag{3.25}$$

Let

d	$= [d_1, d_2, \ldots, d_N]^T$ the desired output vector
w	$= [w_1, w_2, \ldots, w_N]^T$ linear weight vector
N	be the size of the training sample
Φ	denote an N-by-N matrix with element φ_{ji}

$$\Phi = \{\varphi_{ji} \,|(j,i) = 1,2,\ldots,N\} \tag{3.26}$$

This matrix is called the *interpolation matrix* and Equation (3.26) can be written in compact form

$$\Phi w = d \tag{3.27}$$

The unknown weights(w) can be obtained by solving the following linear equation:

$$w = \Phi^+ d \tag{3.28}$$

where Φ^+ is the pseudo-inverse of Φ : $\Phi^+ = (\Phi^T\Phi)^{-1}\Phi^T$

There are many functions to be used in RBF such as

1. Multiquadrics

   $\varphi(r) = (r^2 + c^2)^{\frac{1}{2}}$ for some $c > 0$ and $r \epsilon \mathcal{R}$

2. Inverse multiquadrics

   $\varphi(r) = \frac{1}{(r^2+c^2)^{\frac{1}{2}}}$ for some $c > 0$ and $r \epsilon \mathcal{R}$

3. Gaussian functions

   $\varphi(r) = e^{\left(-\frac{r^2}{2\sigma^2}\right)}$ for some $\sigma > 0$ and $r \epsilon \mathcal{R}$

The multivariate Gaussian function gives two important properties that make the function a proper choice for building an RBF: translation and rotation invariant. The multivariate Gaussian function with these properties is also called *Green's function* of the following form:

$$G(x, x_i) = exp(-\frac{\|x - x_i\|^2}{2\sigma_i^2}) \tag{3.29}$$

RBF Network Structure

The most basic form of RBF network consists of three layers with different roles:

- *input layer* passes the input vectors to the next layers,
- *hidden layer* applies a non-linear transformation function to the input vectors and distributes them to other high-dimensional hidden space,
- *output layer* applies a linear transformation to the activation pattern fed to the input layer.

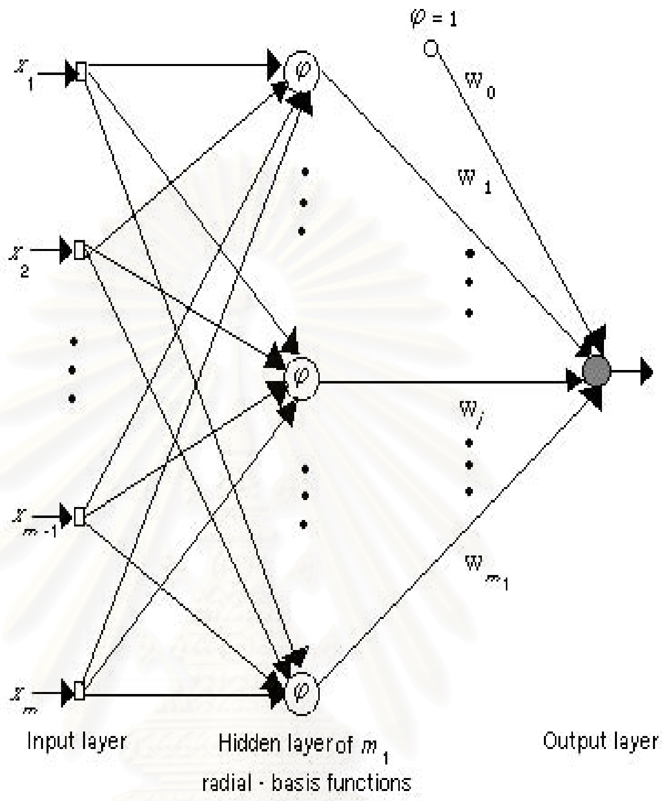


Figure 3.5: Radial-basis function network.

Regularization Networks

Refer to the RBF network structure, the regularization network consists of three layers:

- *input layer* is composed of input nodes that is equal to the dimension m_0 of the input vector x,
- *hidden layer* is composed of nonlinear units that are connected directly to all nodes in the input layer,
- *output layer* consists of a single linear unit fully connected to the hidden layer.

There is one hidden unit for each data point $x_i,\ i = 1, 2, \ldots, N$, where N is the size of the training sample. Green's function is used as the activation function of individual hidden units. Therefore the output of the i^{th} hidden unit is $G(x, x_i)$. The output of the network is a linearly weighted sum of the outputs of the hidden units.

The regularization network models the interpolation function F as a linear superposition (linear weighted sum) of multivariate Gaussian functions whose size is equal to the number of the given sample input N:

$$F(x) = \sum_{i=1}^{N} w_i G(x, x_i) \tag{3.30}$$

or,

$$F(x) = \sum_{i=1}^{N} w_i exp(-\frac{\|x - x_i\|^2}{2\sigma_i^2}) \tag{3.31}$$

where w_i are the weights.

The regularization network provides three important characteristics:

- it is a *universal approximator* in that it can approximate arbitrarily well any multivariate continuous function on a compact set, given a sufficiently large number of units;
- it has the *best approximation property* in that given an unknown nonlinear function there always exists a choice of coefficients that approximates the function better than all other choices; and
- it produces *optimal solutions* that minimize the functional approximation which measures how much the solution deviates from its true value as represented by the training samples.

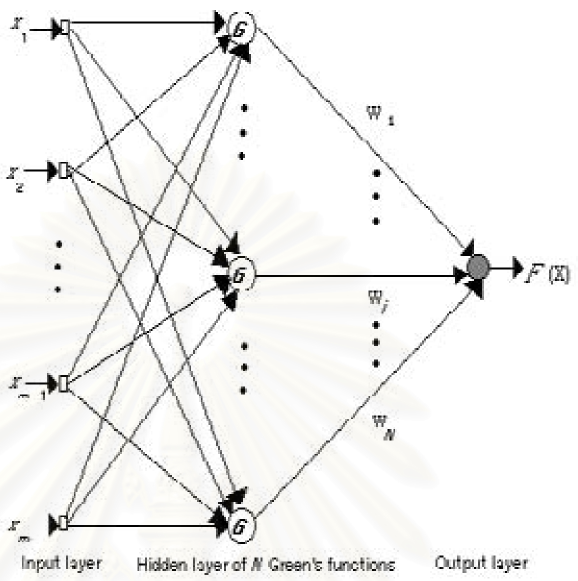


Figure 3.6: Regularization network.

Generalized RBF Networks

In the regularization networks, the number of Green functions is equal to the number of the training examples. This causes computationally inefficient in practice, in the sense that it may require a very large number of basis functions.

In real world practical situations, finding the linear basis function weights needs to invert a very large $N \times N$ matrix which is computationally complex. To overcome this problem, the network complexity needs to be reduced to find a solution that approximates the solution produced by the regularization network. Let $F^*(x)$ be the approximated solution,

$$F^*(x) = \sum_{i=1}^{m_1} w_i \varphi(x) \tag{3.32}$$

where $\{\varphi(x\,|i = 1, 2, \ldots, m_1)\}$ is a new set of basis functions. Typically, the number of basis functions is less than the number of data points ($m_1 \leq N$), and w_i forms a new set of weights. Then,

$$\varphi_i(x) = G(\|x - t_i\|), \;\; i = 1, 2, \ldots, m_1 \tag{3.33}$$

where the set of centers $\{t_i\,|i = 1, 2, \ldots, m_1\}$ is to be determined. Then $F^*(x)$ can be refined as

$$F(x) = \sum_{i}^{m_1} w_i G(x, t_i) \;\; = \;\; \sum_{i=1}^{m_1} w_i G(\|x - t_i\|) \tag{3.34}$$

The RBFN is constructed and trained by the following training algorithm.

RBF Training Algorithm

Initialization: given $(x_e, y_e)\;\; e = 1, 2, \ldots, N$. Consider

- the network structure having a number n of basis functions $\varphi_i, \;\; i = 1, 2, \ldots, n$
- the basis function centers $x_i, \;\; i = 1, 2, \ldots, n$
- the basis function variances $\sigma^2, i = 1, 2, \ldots, n$

Training:

- compute the outputs from each e^{th} sample with the Gaussian basis functions: $\varphi_{ei} = exp(-\frac{\|x_e - x_i\|^2}{2\sigma_i^2})$ at each i^{th} hidden unit, where $i = 1, 2, \ldots, n; \;\; e = 1, 2, \ldots, N$
- calculate the correlation matrix: $\Phi^T\Phi$, and perform the summation: $\Phi^T\Phi + I$
- invert the matrix: $(\Phi^T\Phi + I)^{-1}$

- compute the vector: $\Phi^T y$
- estimate the weight: $w = (\Phi^T \Phi + I)^{-1} \Phi^T y$

CHAPTER IV

FAULT-PRONE PREDICTION

This chapter presents a new approach to predict software faults by means of fuzzy clustering and radial-basis function techniques. Fuzzy subtractive clustering was employed to divide historical and development data into clusters. Next the radial-basis function network was applied to predict software faults that occurred in the component residing in each cluster. In so doing, software faults prediction was accomplished reasonably accurate by Mahaweerawat, et al [2]

4.1 Data and Metric Preparation

The experimental data are collected from 27 business applications by Lanubile [7] and the students at the University of Bari, Italy. The resulting software systems range in size from 1100 to 9400 lines of Pascal source code. There are total 118 components, ranging in size from 60 to 530 lines of code, randomly selected.

The data were tallied and measured using different software metrics, namely, lines of code (LOC), non-comment lines of code ($NCLOC$), Halstead program length (N), Halstead volume (V), McCabe cyclomatic complexity ($V(G)$), Halstead number of unique operands (n_2), Halstead total number of operands (N_2), Henry&Kafura fan-in (fan_{in}), Henry&Kafura fan-out (fan_{out}), Henry&Kafura information flow (IF), and density of comments (DC). These software metrics are derived from the following program parameters:

- McCabe cyclomatic complexity ($V(G)$)

  $V(G) = e - n + 2p$

  where

  G = graph of the flow of program

  e = number of edges

  n = number of nodes

  p = number of unconnected paths of the graph

- fan_{in}: the fan-in of a module M is the number of local flows that terminates at M, plus the number of data structures from which information is retrieved by M.

- fan_{out}: the fan-out of a module M is the number of local flows that emanates from M, plus the number of data structures that is updated by M.

- IF: the information flow is measured by $(fan_{in} \times fan_{out})$

- Halstead's software science

  Length $(N) = N_1 + N_2 = n_1 log_2(n_1) + n_2 log_2(n_2)$

  Volume $(V) = N log_2(n) = N log_2(n_1 + n_2)$

  where

  n_1 = the number of distinct operators that appears in a program

  n_2 = the number of distinct operands that appears in a program

  N_1 = the total occurrences of distinct operator

  N_2 = the total occurrences of distinct operand

- DC: density of comments $= \frac{CLOC}{LOC}$

  where $CLOC$ is the number of comment lines of program text.

The data are divided into training (88 software components) and test (30 software components) sets. A component is represented as a data point consisting of 11 variables for 11 software metrics (this means a point in 11-dimensional space) and each variable value is normalized as:

$$v_{new} = \frac{v_{old} - v_{min}}{v_{max} - v_{min}} \tag{4.1}$$

where v_{new} is new value of the considered variable for the data point, v_{old} is old value of the considered variable, v_{min} is the minimum value of all data point, and the v_{max} is the maximum value of all data point.

4.2 Fault-Prone Predictive Model

Two models were constructed to classify software artifacts as either high-risk which were likely to contain faults, or low-risk which were likely to be fault free. The first model utilized multilayer perceptron (MLP) with back-propagation algorithm, whereas the second model employed RBFN.

4.2.1 The Multilayer Perceptron Network Model

The first model is constructed from multilayer perceptron (MLP) network with back-propagation learning algorithm. The training data set is randomly divided into two parts, i.e., 59 data points for training part and 29 data points for validation part. The training part is used to adjust the weights, whereas the validation part is for error evaluation during training process. The criteria for automatic early stopping in training phase are the PQ_{α} class [45] described in Section 3.5.1 of Chapter 3.

The MLP for this model consists of 11 input nodes in the input layer, 100 nodes in the hidden layer, and 1 output node in the output layer as shown in Figure 4.1. Each hidden node and output node has a bias input which is equal to 1.

The output expected from the model is zero (y = 0) for the fault-free class (low-risk class) and one (y = 1) for the fault-prone class (high-risk class). The learning rate (0.65) and the sigmoid function are used in weight adjustment. The criteria are attained at the 250th training iteration with PQ = 0.0323.

When the training process is complete, the model is re-applied to classify the test data set. The output values so obtained range between 0 and 1 which are indecisive for component classification. Applying a predefined acceptance ratio of 0.55, if the calculated output is greater than or equal to 0.55, the component is a high-risk class. Otherwise, it is a low-risk class. The approach yields a 60% accurate prediction of the test data.

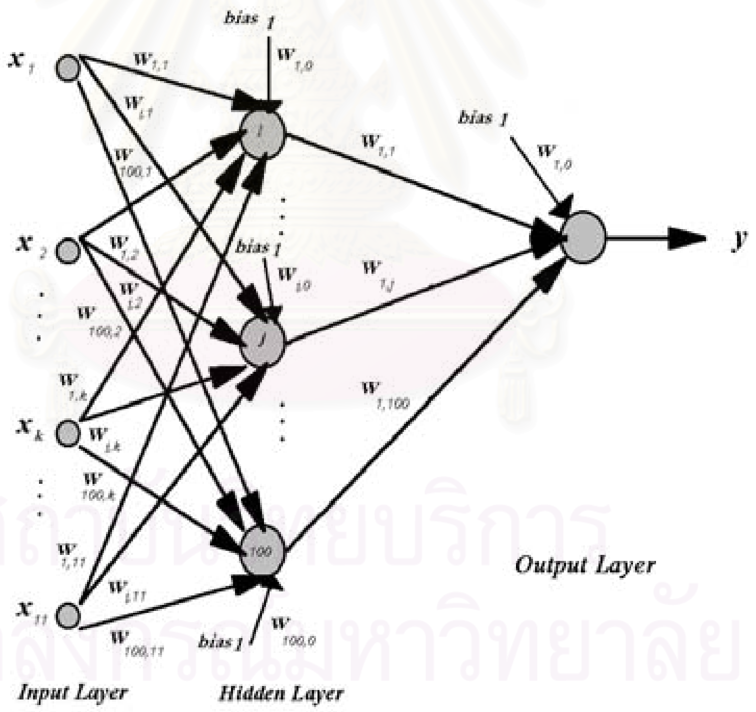


Figure 4.1: Fault-prone predictive model with multilayer perceptron network

4.2.2 The Radial-Basis Function Network Model

In the work of Yuan, et al [20], cluster centers are used to construct fuzzy rules with Sugeno-type. Assuming that the data point is in 2-dimension space (x_1, x_2) and y is a dependent variable or output. Given a cluster center (x_1^*, x_2^*, y^*), fault-proneness can be predicted according to the following premises:

IF x_1 IS CLOSE TO x_1^* AND x_2 IS CLOSE TO x_2^* THEN $y = a_0 + a_1x_1 + a_2x_2$

The relation IS CLOSE TO is implemented as a Gaussian membership function, while the parameter a_j is approximated by linear least squares estimation. This method groups the data points in to proper clusters in which they belong. The output can be calculated using the equation

$$y(x) = \sum_{i=1}^{66} w_i G(\|x - t_i\|) \tag{4.2}$$

From the above equation, it is apparent that Euclidean distance, along with proper weight, serves as the model's IS CLOSE TO relation. Since the calculated output value is a real number close to 0 or 1, it is rounded to the nearest integer (0 for low-risk or 1 for high-risk). Figure 4.2 depicts the structure of this network consisting of 11 input nodes in input layer, 66 hidden nodes and 3 output nodes in hidden and output layers, respectively. This simple adjustment yields an 83% prediction accuracy of the test data.

4.3 Model Evaluation

As both models yield acceptable prediction, it is imperative that their performance be known for subsequent assessment and application. A number of evaluation criteria are considered such as misclassification rate,quality achieved, and verification cost.

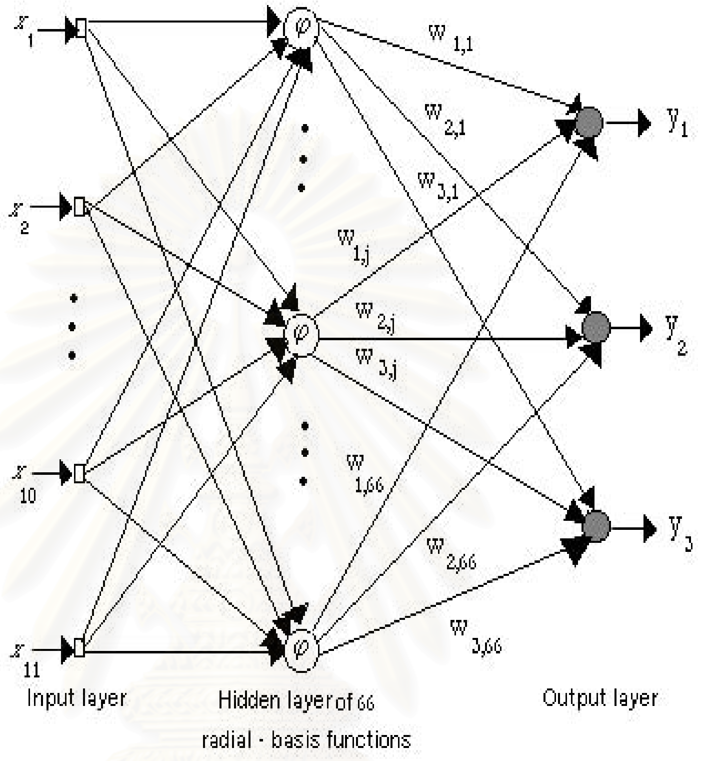


Figure 4.2: The RBFN model

4.3.1 Misclassification rate

There are two types of misclassification errors. A type I error occurs when a high-risk component is classified as low-risk, while a type II denotes a low-risk component being classified as high-risk.

$$\begin{array}{lll} \text{Proportion of Type I} & : & P_1 = \frac{n_{hl}}{n_{tot}} \\ \text{Proportion of Type II} & : & P_2 = \frac{n_{lh}}{n_{tot}} \\ \text{Proportion of Type I + Type II} & : & P_{12} = \frac{n_{hl}+n_{lh}}{n_{tot}} \end{array} \tag{4.3}$$

where n_{hl} is number of type I errors,

n_{lh} is number of type II errors, and

n_{tot} is number of predicted of software components used in the prediction.

4.3.2 Quality achieved

If all the high-risk components are properly classified, all defected will be removed by the extra verification, and perfect quality will be achieved.

The quality completeness measure is carried out according to the equation

$$C = \frac{n_{hh}}{n_{rh}} \tag{4.4}$$

where n_{hh} is number of faulty components that have been actually classified as such by the model, and

n_{rh} is total faulty components.

4.3.3 Verification cost

Two indicators are used to measure the verification cost, inspection and wasted inspection. Inspection (I) measures the overall cost by considering the percentage of components that should be verified. Waste inspection (WI) is the percentage of components that do not contain faults but have actually been verified because they were incorrectly classified as faulty.

$$\begin{aligned} \mathrm{I} &= \tfrac{n_{ph}}{n_{tot}} \\ \mathrm{WI} &= \tfrac{n_{lh}}{n_{ph}} \end{aligned} \tag{4.5}$$

Table 4.1: Results from model evaluations

Criterion	**Fault predictive model**	
	MLP model	**RBF model**
Misclassification rate	$P_1 = 0$	$P_1 = 6.67$
	$P_2 = 40.00$	$P_2 = 10.00$
	$P_{12} = 40.00$	$P_{12} = 16.67$
Quality achieved	$C = 100$	$C = 88.24$
Verification cost	$I = 96.67$	$I = 60.00$
	$WI = 41.38$	$WI = 16.67$

where n_{ph} is number of components that have been classified as high-risk class,

n_{lh} is number of low-risk components that have been classified as high-risk class, and

n_{tot} is number of all components.

The results of model evaluations can be interpreted in Table 4.1:

From the above results, misclassification rate of the fault-prone MLP model is higher than that of the fault-prone RBF model. The rate at which a low-risk component is misclassified as a high-risk component is high in the MLP model, while there is no high-risk component misclassified as low-risk class. Although the completeness of the MLP model is higher than the RBF model, it expends more wasted cost of inspection. The more important aspect, perhaps, is that the RBF model can accurately predict fault-prone component up to 83% as oppose to 60% by the MLP model.

CHAPTER V

FAULT TYPE PREDICTION

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 chapter, a new model [4, 5], called MASP, for predicting and identifying of faults in object-oriented software systems is introduced. In particular, faults due to the use of inheritance 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.

5.1 Fault Categories

Inheritance provides many benefits in creativity, efficiency and reuse but they can cause a number of anomalies and faults [35]. This study focuses on five fault types caused by the use of inheritance as introduced in [35] and are summarized in Table 5.1. The following descriptions of fault types use the concepts of class extension and refinement defined in [35] as follows. A class *extends* its parent classes if it introduces a new method name and does not override any methods in its ancestor classes. A class *refines* the parent classes if it provides new behaviors not presented in the overridden method, does not call the overridden method, and its behavior is semantically consistent with that of the overridden method.

State definition anomaly (SDA). An anomaly will exist if the refining methods implemented in the descendant that provides definitions for the inherited state variables. The definitions must be inconsistent with those in the overridden method or any extension method that is called by a refining method. Consequently, the inherited variables so defined are inconsistent with those of the ancestor's current state.

State definition inconsistency due to state variable hiding (SDIH). This fault is due to the introduction of an indiscriminately named local state variable v. If a local variable is introduced to a class definition where the name of variable is the same as an inherited variable from its ancestor, the inherited variable is hidden from the scope of the descendant as the variable is implicitly referred causing faults to occur.

State defined incorrectly (SDI). When an overriding method defines the same state variable v as defined in the overridden method, but the computation performed by the overriding method is not semantically equivalent to the computation of the overridden method with respect to v, then the fault will result.

Indirect inconsistent state definition (IISD). A descendant class D adds an extension method *e()* that defines an inherited state variable v. A method *m()* in its ancestor class T cannot directly call the extension method *e()* unless the descendant class D inherit the method *m()*. Then the inherited method *D::m()* can

Table 5.1: Fault and anomalies due to inheritance

Acronym	**Fault/Anomaly**
SDA	State Definition Anomaly (possible post-condition violation)
SDIH	State Definition Inconsistency (due to state variable hiding)
SDI	State Definition Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
SVA	State Visibility Anomaly

call the method *e()*. An IISD will occur if the extension method *e()* defines a state variable v of the ancestor class T that yields the state of the ancestor class incorrectly.

State visibility anomaly (SVA). The state variables v in an ancestor class A are declared private and subsequently defined by a polymorphic method *A::m()*, i.e., *A::v*. Suppose that B is a descendant of A, and C of B. Both class B and C provide an overriding definition of *A::m()*. Class C has to call *A::m()* to modify v. The SVA will be introduced if class C calls *B::m()* instead of *A::m()* and class B has refined the method *m()* and yields inconsistent state with respect to v.

5.2 Software Metrics and Parameters

In general, software metrics provide quantitative descriptions of certain characteristics of software products and processes. Code metrics, as a specific type of product metrics, provide such descriptions for elements of software systems, e.g., classes in case of object-oriented systems. Table 5.2 lists a set of metrics [24] provided by the software tool "Understand for C++" [46] are employed to measure the above software faults.

Table 5.2: 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)

The number of syntactic fault patterns [3] in object-oriented programs according to the fault types in Section 5.1 has been used as additional parameters. Table 5.3 summarizes those patterns whose definitions are as follows:

- **Extension method Calls another Extension method (ECE).** Descendant classes that use ECE mechanism can cause SDA anomalies if the called extension method *c* defines inherited state variables or calls inherited methods that defines inherited state variables. Method *c* can yield an anomaly if a method that is subsequently called depends in some way on the state defined by *c*.

- **Extension method Calls Inherited methods (ECI).** An SDA anomaly will exist if the inherited method *i* which defines and uses a state variable (in the ancestor's context) is called out of sequence with respect to the current state by an extension method. An SVA fault can also appear due to ECI if a state variables *v* in an ancestor class *A* are declared private and subsequently defined by a polymorphic method *A::m()*, i.e., *A::v.* Suppose that *B* is a descendant of *A*, and *C* of *B*. Both class *B* and *C* provide an overriding definition of *A::m().* Class *C* has to call *A::m()* to modify *v.* Then class *C* calls *B::m()* instead of *A::m()* and class *B* has refined the method *m()* and yields inconsistent state with respect to *v.* If an extension method *c* of class *C* calls the inherited method *C::m()* but the method *B::m()* is refined in some way that is inconsistent with *A::m()*, then an ECI fault occurs.

- **Extension method Calls Refining method (ECR).** If a refining method *r* defines inherited variables, an SDA anomaly can occur if a subsequently called method depends upon the ancestor's state in some way that has been affected by *r.* In addition, a fault can exist if *r* defines the state variable incorrectly or uses the state variable and is called out of sequence with respect to the current state of the ancestor.

- **Extension method Defines Inherited state Variable (EDIV).** An SDA anomaly will be introduced if the extension method defines an inherited variable v at a time that is inconsistent with the current state of the ancestor. Furthermore, an SDI fault can appear if the definition given to v is not consistent with how the variable is defined by ancestor methods.

- **Refining method Calls Extension method (RCE).** A refining method manifests SDA by failing to define the same set of the ancestor's state variables as the overridden method does. Although the refining method defines the right state variables, it can cause an SDI fault if the refining method defines them incorrectly. An IISD can occur if the refining method r calls one of the descendant's extension methods.

- **Refining method Calls other Inherited method (RCI).** An SDA anomaly can exist if a refining method that calls an inherited method i (instead of the overridden method o) and the inherited method i defines different set of state variables as the overridden method o does. However, if i defines the same set of state variables as o does, it can cause an SDI fault if the semantics of the resulting definition are different. An SVA fault can also appear due to RCI. Given the state variables in an ancestor class A are declared private, and a method *A::m()* defines *A::v*. Suppose B is a descendant of A, and C of B. The descendant class C calls the inherited method *C::m()* which, in turns, is inherited from *B::m()* to modify v. If a refining method r of class C calls the inherited method *C::m()* but the method *B::m()* is refined in some way that is inconsistent with *A::m()*, then an SVA fault occurs.

- **Refining method Calls another Refining method (RCR).** The effects of a refining method calling another refining method are similar to refining method calling an extension method (RCI). It can cause both SDA anomalies and SDI faults.

- **Refining method Calls Overridden Method (RCOM).** If a refining method r calls the overridden method o and defines additional state variables not defined by o or redefines those defined by o, then SDA anomalies and SDI faults occur.

- **Refining method Defines/Uses Inherited state Variable (RDIV/RUIV).** RDIV can causes both SDA anomalies and SDI faults. If a refining method does not define the same state variables as the overridden method, then an SDA anomaly will occur. If the refining method defines the same state variables as the overridden method but in a manner that is inconsistent with how the overridden method defines, then an SDI fault will exist. Moreover, RUIV can introduce an SDIH anomaly if the refining method r declares a local state variable v whose name is identical to one that is inherited and r uses v to define an inherited state variable.

Besides the number of the patterns and software metrics described in [3, 34], additional parameters are defined for use in 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.

Table 5.3: 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

- **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 its 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 the 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 SDA and SDI fault 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 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 but not consistent with the original method in the indirect base class, then an SVA fault appears.

The parametric measurements are categorized according to the five fault types shown in Table 5.4.

5.3 Fault Analysis

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

A faultiness predictive model is constructed based on software characteristics, such as fan-in/fan-out, modularity, and cohesion, that are measured by selected software metrics

Table 5.4: Fault/anomaly types identified by syntactic patterns and parameters.

Pattern/	**Fault Type**				
Parameter	**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

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 utilized to build the predictive model. In a preliminary investigation [2], 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 Figure 5.1.

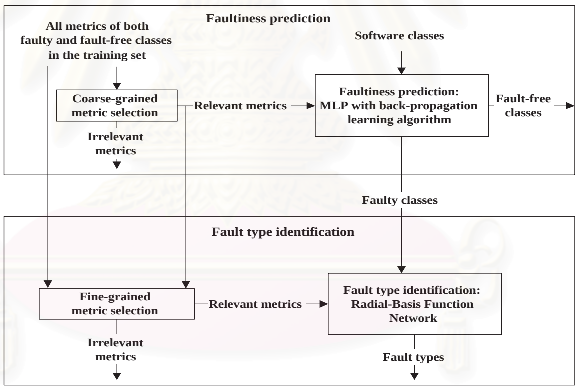

Faultiness prediction
All metrics of both faulty and fault-free classes in the training set
Software classes
Coarse-grained metric selection
Relevant metrics
Faultiness prediction: MLP with back-propagation learning algorithm
Fault-free classes
Irrelevant metrics
Faulty classes
Fault type identification
Fine-grained metric selection
Relevant metrics
Fault type identification: Radial-Basis Function Network
Irrelevant metrics
Fault types

Figure 5.1: Diagram of MASP fault identification model construction.

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 then applied to extract faulty classes using MLP 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 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.

5.4 Faultiness Prediction–A Coarse-Grained Approach

A coarse-grained approach employs selected metrics from a coarse-grained metric selection algorithm to construct a faultiness predictive model.

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 varied between 100 and 500 lines of code. Such combinations of experimental data provided 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 5.1 inserted according to syntactic patterns in [3]. All faulty and fault-free samples were measured with 60 software metrics and fault parameters given in [3, 46].

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 5.5 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.

Table 5.5: Training and test data sets.

Fault	**A**		**B**		**C**	
Category	**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

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 [22, 25, 26, 29, 30] 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 statistical and mathematical methods are black box which cannot explain the reasoning behind the metric selection [23], 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 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 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 [22,25,26,29,30], i.e., univariate logistic regression, multivariate logistic regression, PCA, and an unsupervised method presented in [47]. 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. Performance of MASP model was compared with other approaches. The objective was to fine tune the proposed model to correctly classify the data points into fault-free and fault groups. The structure of faultiness predictive model consisted 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.

Table 5.6: 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%

The expected output value computed from the output node of the 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 was a fault-free class. The comparative results of the experiments are depicted in Table 5.6. Each faultiness predictive model was built from the training set A and re-applied to the test set A, both training and test sets 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 5.7.

The highest correctness percentage was accomplished by the proposed model and was subsequently evaluated through some measurement criteria [7] as follows:

Table 5.7: 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

- **Type I error (T1):** This error occurs when a faulty class is classified as fault-free; T1 = 2.81%

- **Type II 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 verification to see if they are indeed faulty; 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%

5.5 Fault Type Identification–A Fine-Grained Approach

A fine-grained metric selection algorithm is proposed. The algorithm is based on 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.

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 a 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 5.8 that the thirty-five fine-grained selected metrics were composed of the same eleven metrics obtained from the coarse-grained algorithm in Section 5.4 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', '00100', '00010', and '00001', representing SDIH, IISD, SVA, SDA, and SDI faults, respectively.

During the experiment, training data were used to generate weights between the hidden layer and the output layer. If the network yielded low accuracy, the number

Table 5.8: 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

Table 5.9: Results from applying the fault type predictive model to predict faulty classes.

Fault	**Predicted Fault Type**				
Category	**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

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 lower accuracy was that some fault-free classes were incorrectly classified as faulty classes by the faultiness predictive model presented in Section 5.5. The results shown in Table 5.9 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 metric among all 35 that dominates fault type of a given hidden node, 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 the output nodes.
3. Identify the set of classes from training data where the selected fault is originated.
4. For each metric, calculate the difference between metric values of a training class and a hidden node (each of which contains 35 metrics).

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

   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 same 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.

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

Figure 5.2 and Figure 5.3 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 Figure 5.3, 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, Figure 5.2 shows that this metric has less effect on IISD fault.

Figure 5.4 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 Figure 5.5. 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) [46] has on every fault type.

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

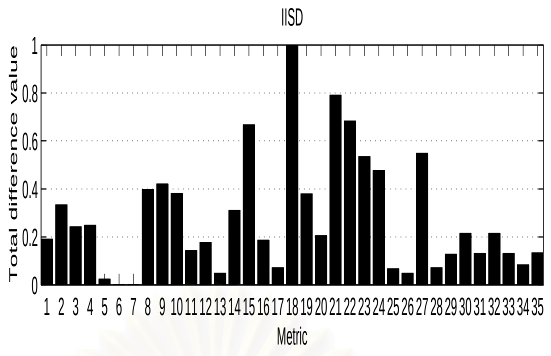


Figure 5.2: The total difference of each metric between hidden nodes and training classes having IISD fault.

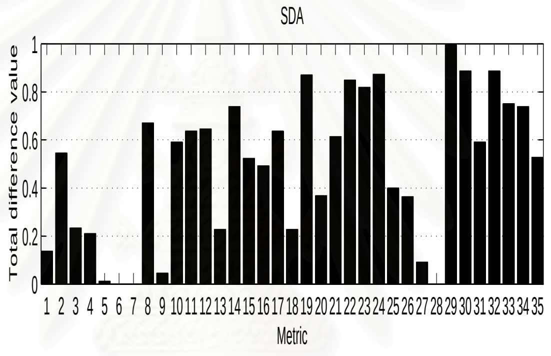


Figure 5.3: The total difference of each metric between hidden nodes and training classes having SDA fault.

Table 5.10: 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%

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.

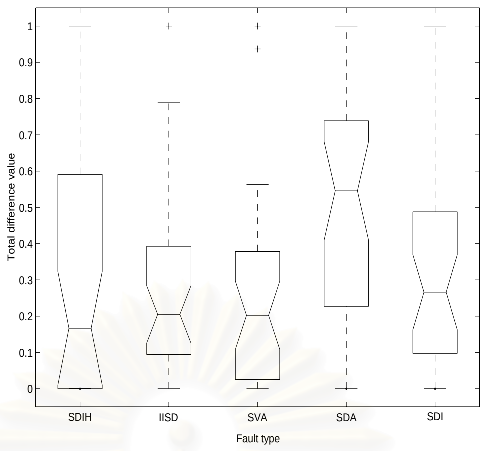


Figure 5.4: The total difference of all metrics between hidden nodes and training classes for each fault type.

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 5.10. 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 contributing fault types can also be identified by MASP's pair-wise metric comparison algorithm in Section 5.5. 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. It is envisioned that some forms of fine grained metric preprocessing for each particular fault type should be car-

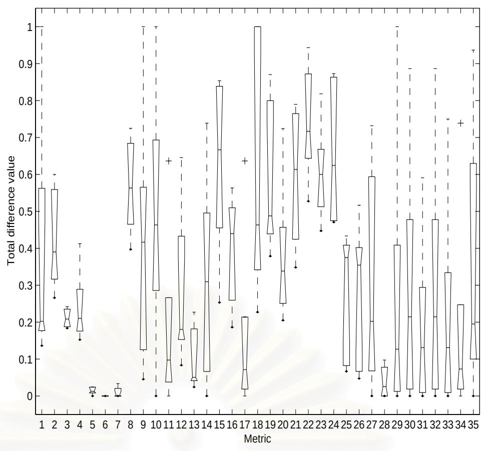


Figure 5.5: The total difference of all metrics between hidden nodes and training classes for all fault types.

ried out to alleviate the aforementioned caveats (as shown in Table 5.9 and 5.10) and consequently reduce the costs incurred.

CHAPTER VI

DYNAMIC FAULT PREDICTION

Software faults have been widely studied in both procedural and object-oriented programming, where most researches utilize static metrics obtained from source code to predict the faultiness. However, there exists faults which occur at run time and may not be detected by static metrics. In this chapter, dynamic fault analysis is employed to probe those faults in object-oriented programming. Each object-oriented program is represented as graphs, along with some fault metrics extracted from those graphs are proposed. Fault prediction process is performed based on the fault metrics so obtained using neural network techniques. The cause and location of predicted faults are then determined from the proposed graphs with the help of sensitivity analysis technique.

6.1 Software Model

Investigation of dynamic faults calls for an in-depth dichotomy of program behavior. In this study, a graphical model to denote an object-oriented program in a series of graphs has been established. Each graph contains different nodes and edges corresponding to what they represent such as class, method, membership relation, etc. These representative graphs have general definitions that may encompass additional information given in [37, 48] as follows:

Definition 6.1: A digraph G is a pair (V, E), where V is a finite, nonempty set of vertices, and $E \subseteq V \times V$ is a set of distinct ordered pairs V. Elements of E are called directed edges. For an edge $e \in E$ from a vertex v_1 to a vertex v_2, v_1 is the tail and called the initial vertex of e, denoted as $IV(e)$, and v_2 is the head and called the terminal vertex of e, denoted as $TV(e)$.

Definition 6.2: A multi-digraph is an $n+1$ tuple $(V,\ E_1,\ E_2, \ldots,\ E_n)$ such that for all i, $1 \leq i \leq n$, (V, E_i) is a digraph. A path in the graph is a sequence of edges $(e_1,\ e_2, \ldots,\ e_k)$ such that $TV(e_i) = IV(e_{i+1})$, for $1 \leq i \leq k-1$, and $e_j \in \bigcup_{p=1}^{n} E_p$, for $1 \leq j \leq k$. Let $v_I = IV(e_1)$ and $v_T = TV(e_j)$. The connection is called a path from v_I to v_T, denoted by $v_I \longrightarrow v_T$ for short.

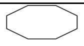
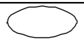
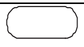
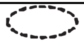
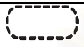
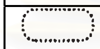
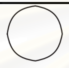
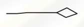
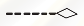
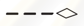
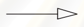
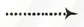
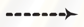
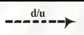
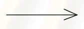
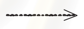
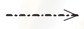
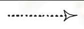
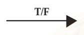
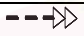
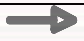
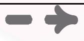
Definition 6.3: Let $e_1,\ e_2,\ \ldots,$ and e_k be a set of edges in a multi-digraph $G(V,\ E_1,\ E_2, \ldots,\ E_n)$, v_I be the initial vertex of e_1, and v_T be the terminal vertex of e_k. $(e_1,\ e_2,\ \ldots,\ e_k)$ is a path in the graph if and only if the terminal vertex of e_i is the initial vertex of e_{i+1} for $1 \leq i \leq k-1$. Denote the path from v_I to v_T as $v_I \longrightarrow v_T$. For a path $v_p \longrightarrow v_q = (e_1, e_2, \ldots, e_n) V_p \xrightarrow[E_x \cup E_y \cup \ldots \cup E_z]{} V_q$ means that $\forall j, 1 \leq j \leq n, e_j \in E_x \cup E_y \cup \ldots \cup E_z$.

Definition 6.4: A tagged multi-digraph is a tuple $(V, (\ E_1,\ E_2, \ldots,\ E_n),\ T)$ such that $(V,\ E_1,\ E_2, \ldots,\ E_n)$ is a multi-digraph, where

1. V is a finite set of vertices,
2. $E_i \subseteq V \times V$ is a finite set of directed edges, for $1 \leq i \leq n$, and
3. T is a finite set of vertex tags; $T = \bigcup_{i=1}^{m=|V|} T(X_i)$, where $T(X_i)$ is a set of vertex tags associated with vertex X_i, $X_i \in V$, $i = \{1, 2, \ldots, m\}$.

An object-oriented program analysis approach is introduced, encompassing five aspects, namely, program structure, class inheritance, association, and control flow. For

Table 6.1: Symbols in graphs

Symbol	Abbreviation	Description
	V_c	Class
	V_a	Attribute
	V_m	Method
	V_{ari}	Referred inherited attribute
	V_{mri}	Referred inherited method
<OP$_{si}$,OP$_{bi}$>	T(X)	Vertex tag
	V_{gf}	Global function
	V_{sec}	Source code section
	E_{pri}	Private membership
	E_{pro}	Protected membership
	E_{pub}	Public membership
	E_h	Class inheritance
	E_{mh}	Member inheritance
	E_m	Method invocation
	E_{da}/E_{ua}	Attribute access
	E_{hpri}	Private inheritance
	E_{hpro}	Protected inheritance
	E_{hpub}	Public inheritance
	E_{fa}	Friend association
	E_{ctl}	Control flow
	E_{inv}	involving
	E_{IHp}	Inheritance path
	E_{ASSp}	Association path

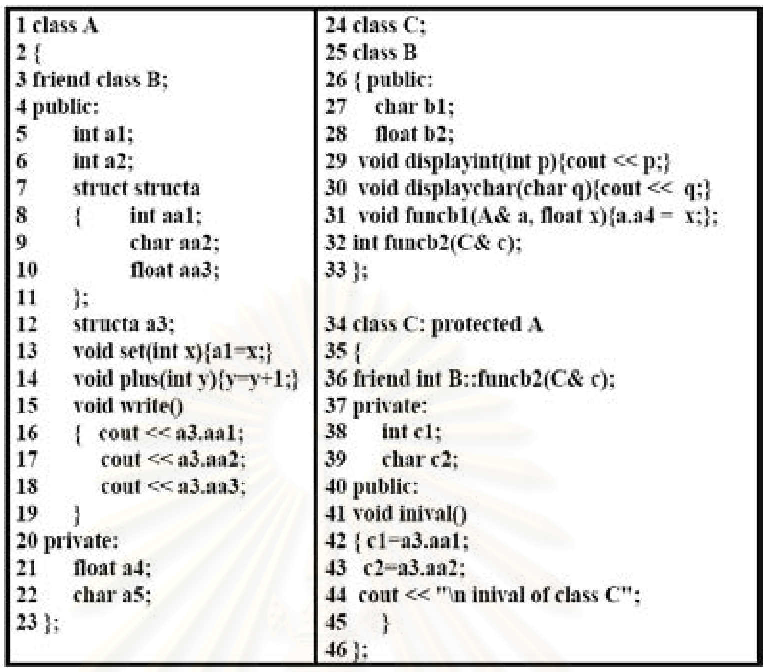
clarity, a list of symbols and graphs used in this article is given in Table 6.1.

6.1.1 Program Structure

An object-oriented program structure can be represented by a Program Structure Graph (PSG) [37]. A PSG is a multi-digraph, where vertices represent classes, methods, and attributes; multiple edge sets represent class inheritance, public-/protected-/private-memberships, declarations, attribute access, and method invocation. To apply PSG to this study, some definitions on inherited attribute and method are established and the original PSG descriptions are redefined as follows:

Definition 6.5: Let P be an object-oriented program. A PSG of P is defined as $G_{PSG}(P) = (V, E)$, where

1. $V = V_c \cup V_m \cup V_a \cup V_{mri} \cup V_{ari}$ such that
   - V_c is a set of vertices representing classes,
   - V_m is a set of vertices representing methods,
   - V_a is a set of vertices representing attributes,
   - V_{mri} is a set of vertices representing inherited methods which are redefined or referred in the class, and
   - V_{ari} is a set of vertices representing inherited attributes which are redefined or referred in the class.
2. $E = (E_h,\ E_{pub},\ E_{pro},\ E_{pri},\ E_m,\ E_{da},\ E_{ua})$ such that
   - $E_h \subseteq V_c \times V_c$ is a set of edges from a class to its immediate subclass representing class inheritance,
   - $E_{pub} \subseteq (V_m \cup V_a \cup V_{mri} \cup V_{ari}) \times V_c$ is a set of edges from an attribute or a method to its defining class representing public-membership,
   - $E_{pro} \subseteq (V_m \cup V_a \cup V_{mri} \cup V_{ari}) \times V_c$ is a set of edges from an attribute or a method to its defining class representing protected-membership,
   - $E_{pri} \subseteq (V_m \cup V_a \cup V_{mri} \cup V_{ari}) \times V_c$ is a set of edges from an attribute or a method to its defining class representing private-membership,
   - $E_m \subseteq (V_m \cup V_{mri} \times (V_m \cup V_{mri})$ is a set of edges from a source method to another method invoking the source method,



```
class A
{
friend class B;
public:
      int a1;
      int a2;
      struct structa
      {       int aa1;
              char aa2;
              float aa3;
      };
      structa a3;
      void set(int x){a1=x;}
      void plus(int y){y=y+1;}
      void write()
      {  cout << a3.aa1;
         cout << a3.aa2;
         cout << a3.aa3;
      }
private:
      float a4;
      char a5;
};
```

```
class C;
class B
{ public:
    char b1;
    float b2;
 void displayint(int p){cout << p;}
 void displaychar(char q){cout <<  q;}
 void funcb1(A& a, float x){a.a4 =  x;};
int funcb2(C& c);
};

class C: protected A
{
friend int B::funcb2(C& c);
private:
     int c1;
     char c2;
public:
void inival()
{ c1=a3.aa1;
 c2=a3.aa2;
 cout << "\n inival of class C";
   }
};
```

Figure 6.1: Class A, B, and C

- $E_{da} \subseteq (V_a \cup V_{ari}) \times (V_m \cup V_{mri})$ is a set of edges from an attribute to a method defining the attribute value, and
- $E_{ua} \subseteq (V_a \cup V_{ari}) \times (V_m \cup V_{mri})$ is a set of edges from an attribute to a method using the attribute value.

Figure 6.1 and 6.2 show example programs containing six classes. A PSG can be constructed from the example programs as shown in Figure 6.3. All attribute accesses and method invocations of a class are also demonstrated in attribute access graphs and method invocation graphs shown in Figure 6.4 through 6.10.

6.1.2 Class Inheritance

An inheritance relationship between one class and its subclasses means that the subclasses can possess members of the class and be identified according to declarations of

```
class E;
class D: public A, public B
{
friend void func1(D&,E&);
public:
       float d1;
       char d2;
        void setd1()
        { b2 = b2*0.1;
          d1 = b2 + a1;
        }
        void display()
        { displayint(d1);
          displaychar(d2);
        }
protected:
 void set(int x){a1=x+1;}
};

class E
{
friend void func1(D&,E&);
public:

       int e2;
       void plusval()
       {
         e2 = e2+e4;
       }

private:
       int e4;
};
```

```
static class F: private D
{
 public:
 void compute()
 { A::set(2);
   f1 = (a1 + b2) * d1;
 }
void write()
 {  cout <<"\n a1: ";
     cout << a1;
     cout <<"\n b2: ";
     cout << b2;
     cout <<"\n d1: ";
     cout << d1;
     cout <<"\n f1: ";
     cout << f1;
 }
 void setd1(float d){F::d1 = d;}
 void setb2(float b){F::b2 = b;}
 F();

private:
      float f1;

};
F::F(){};
int B::funcb2(C& c){return c.c1;}
void func1(D& d,E&e){d.set(e.e4);}
```

Figure 6.2: Class D, E, and F

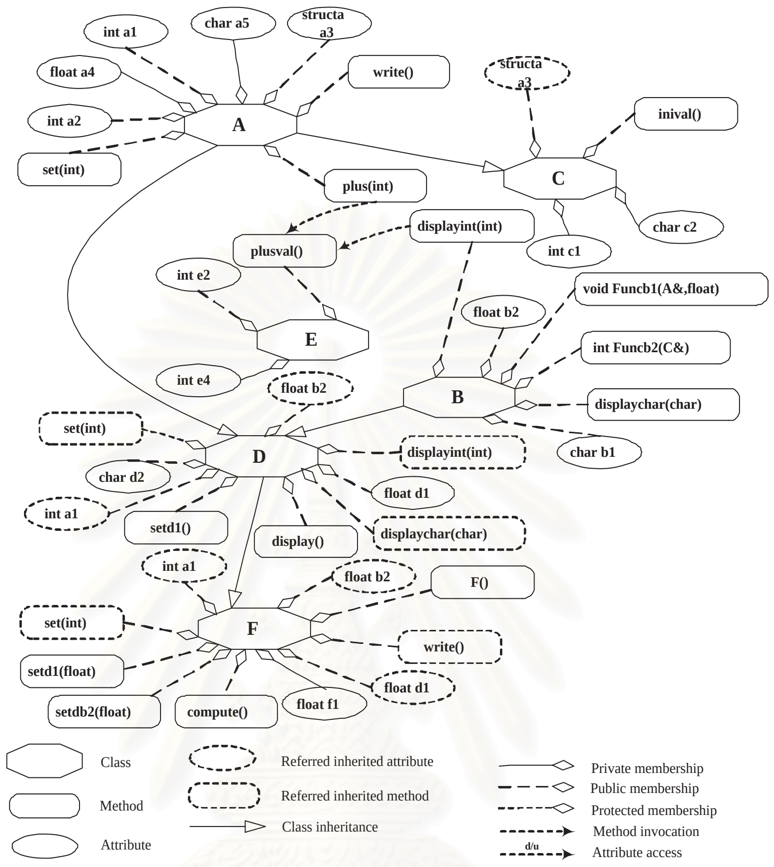


Figure 6.3: Program structure graph

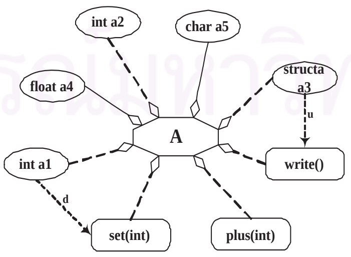


Figure 6.4: Attribute access graph of class A

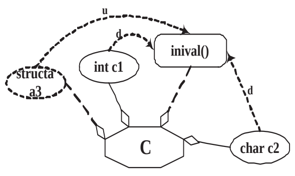


Figure 6.5: Attribute access graph of class C

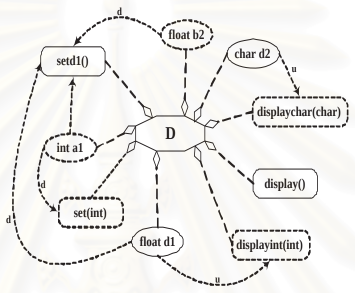


Figure 6.6: Attribute access graph of class D

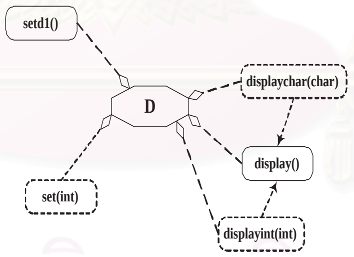


Figure 6.7: Method invocation graph of class D

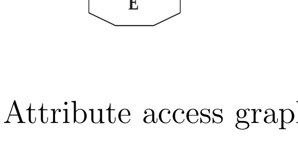


Figure 6.8: Attribute access graph of class E

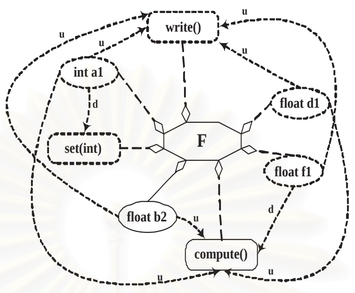


Figure 6.9: Attribute access graph of class F

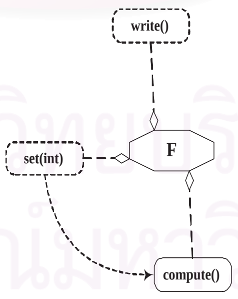


Figure 6.10: Method invocation graph of class F

the following forms:

class $ClassP$: $ClassQ$

class $ClassP$: `public` $ClassQ$

class $ClassP$: `protected` $ClassQ$

class $ClassP$: `private` $ClassQ$

An Inheritance Flow Graph (IFG) [49] is employed to represent inheritance relationship among classes and to identify flow operations which are either defined or used in association with other members of the class. The notion is then applied in this research with the help of some added and redefined definitions as follows:

Definition 6.6: Type and structure of an attribute are denoted in its signature but does not contain a body.

Definition 6.7: Type, number, and order of in/out parameters of a method are denoted in the signature of the method while the body of the method holds the statements or other implementation details.

Definition 6.8: An operation on a member of a class is either a signature-inheritance define (D_{si}) or signature-inheritance use (U_{si}).

1. a D_{si} on a member means that the signature of the member is originated in the class,
2. a U_{si} on a member means that the signature of the member is inherited from a superclass, i.e., the class processes the signature without defining it.

Definition 6.9: An operation on the body of a member in a class is either body-inheritance define (D_{bi}), body-inheritance use (U_{bi}), or null (N_{bi}).

1. a D_{bi} on a member means that the body of the member is newly defined or redefined in the class,

2. a U_{bi} on a member means that the body of the member exists, no new definition is specified in the class,

3. an N_{bi} on a member means that neither D_{bi} nor U_{bi} constitutes the body of the member, i.e., the body does not exist.

A pair of operations on a member of a class in inheritance flow are a combination of $\{D_{si}, D_{ui}\} \times \{D_{bi}, U_{bi}, N_{bi}\}$. There are six combinations, but only five of them are applied. The operation pair (D_{si}, U_{bi}) are excluded because no member can redefine the signature while its body is kept unchanged.

Definition 6.10: Let P be an object-oriented program. An Inheritance Flow Graph (IFG) of P is defined as $G_{IFG}(P) = (V, E, T)$, where (V, E, T) are tagged multi-digraphs, and

1. $V = V_c \cup V_m \cup V_a \cup V_{mri} \cup V_{ari}$ as defined earlier in definition 6.5

2. $E = (E_{hpub}, E_{hpro}, E_{hpri}, E_{pub}, E_{pro}, E_{pri}, E_{mh}, E_m, E_{da}, E_{ua})$ such that

   - $E_{hpub} \subseteq V_c \times V_c$ is a set of edges from a class to its immediate subclass representing public inheritance,

   - $E_{hpro} \subseteq V_c \times V_c$ is a set of edges from a class to its immediate subclass representing protected inheritance,

   - $E_{hpri} \subseteq V_c \times V_c$ is a set of edges from a class to its immediate subclass representing private inheritance,

- $E_{mh} \subseteq ((V_a \cup V_{ari}) \times V_{ari}) \cup ((V_m \cup V_{mri}) \times V_{mri})$ is a set of edges from an attribute (referred inherited attribute) or a method (referred inherited method) defining signature/body inheritance of a superclass to a referred inherited attribute or method of a subclass, and
- E_{pub}, E_{pro}, E_{pri}, E_m, E_{da}, E_{ua} as defined earlier in definition 6.5

3. $T = \bigcup_{i=1}^{m} T(X_i)$, where $T(X_i)$ is a pair of vertex tags. Denote $\langle a, b\rangle$ as a pair of flow operations on the signature and body of a member associating with X_i, where $X_i \in V_a \cup V_m \cup V_{ari} \cup V_{mri}$, $i = \{1, 2, \ldots, m\}$, $m = |V_a \cup V_m \cup V_{ari} \cup V_{mri}|$

An implicit member is a member inherited from a superclass without any change, denoted by a pair of vertex tags $\langle U_{si}, U_{bi}\rangle$ or $\langle U_{si}, N_{bi}\rangle$. This member is represented as V_{ari} or V_{mri} in an IFG if it is referred in a class. Figure 6.11 illustrates class inheritance from all classes of the example program in Figure 6.1 and 6.2.

Definition 6.11: Let q_1 and q_2 be two class vertices in an IFG. A flow path from vertex q_1 to vertex q_2 is an inheritance flow path, denoted as $q_1 \xrightarrow[IHF]{} q_2$, if and only if one of the following holds:

1. $q_1 \rightarrow q_2 \in E_{hpub} \cup E_{hpro} \cup E_{pri}$, or
2. $\exists \alpha$, $\alpha \in V_c$ such that $q_1 \rightarrow \alpha \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$ and $\alpha \xrightarrow[IHF]{} q_2$.

The inheritance flow graph and inheritance flow path are depicted in Figure 6.11 and 6.12, respectively.

6.1.3 Association

An association relationship between two classes means that one class members can be used in the other's members. The "use" class sends a message or its instance to an

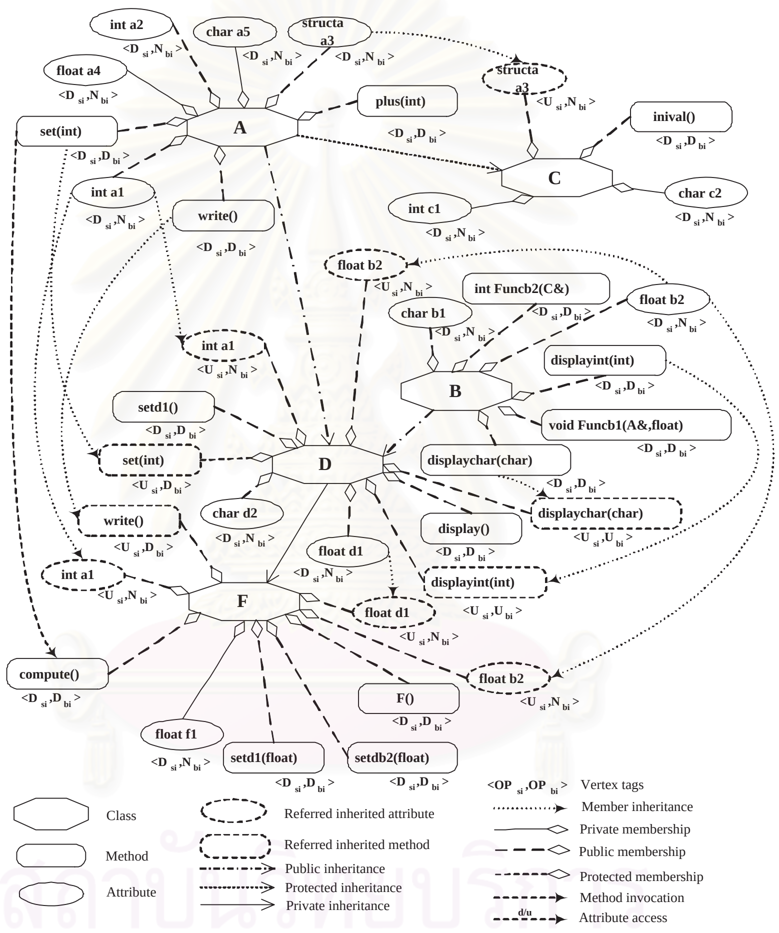


Figure 6.11: Inheritance flow graph

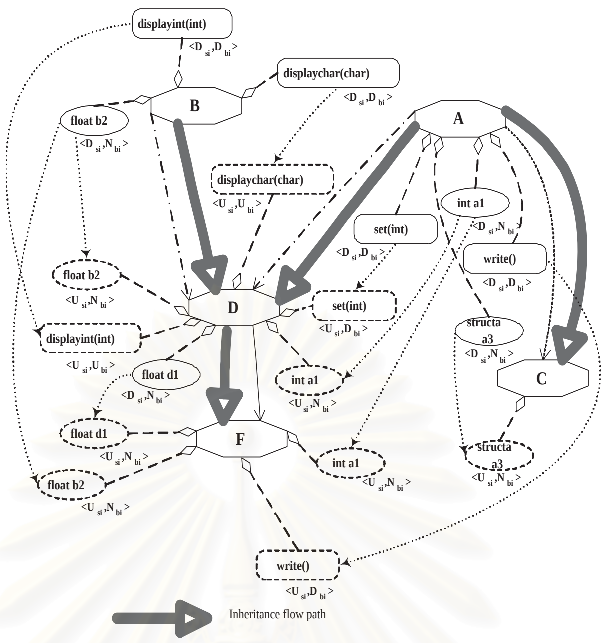


Figure 6.12: Inheritance flow path

instance or a member function of the "used" class [37, 38]. There are four types of association relationships [39]:

- Friend member function association is identified according to declarations of the following form

```
class ClassP {
  //...
  friend  return_type ClassQ::f(...);
  //...
};
```

- Friend class association is identified according to declarations of the following form

```
class ClassP {
  // ...
  friend class ClassQ;
  // ...
};
```

- Friend operation association is the association between classes through a global function. For example,

```
class date;
class time {
  // ...
  friend char *timedate(time, date);
  // ...
};
class date {
  // ...
  friend char *timedate(time, date);
  // ...
};
char *timedate(time t, date d) {...};
```

- Ordinary association is the association established through parameter passing of an instance of one class to a member function of another class.

In the ordinary association, execution of a method in an instance of one class might send a message to an instance of another class to invoke the designated method. A method might be invoked by a number of other methods. The method invocation sequence along with association relationships form the association flow. Thus, an association flow path is a sequence of association relationships. The corresponding flow operations on the members of associated classes along the path are defined as follows [37]:

Definition 6.15: An operation on a member of a class is a define association (D_{as}) or a use association (U_{as}).

1. A D_{as} on a member means that the class owns the member,
2. A U_{as} on a member means that the class contains a message that might access or invoke the member.

A member with the operation D_{as} means that the class explicitly defines or inherits the member from other classes. A U_{as} on a member means that the member might be invoked by some messages associating with the class.

Definition 6.16: Let P be an object-oriented program. An Association Graph (ASG) of P is defined as $G_{ASG}(P) = (V, E)$, where (V, E) is a multi-digraph, and

1. $V = V_c \cup V_m \cup V_a \cup V_{ari} \cup V_{mri} \cup V_{gf}$ such that
   - V_{gf} is a set of vertices representing global functions,
   - V_c, V_m, V_a, V_{ari}, V_{mri} as defined earlier in definition 6.5, and
2. $E = (E_{pub}, E_{pro}, E_{pri}, E_m, E_{da}, E_{ua}, E_{fa})$ such that
   - $E_{fa} \subseteq V_c \times (V_c \cup V_m \cup V_{gf})$ is a set of edges from a class to a class, a method, or a global function,
   - E_{pub}, E_{pro}, E_{pri} as defined earlier in definition 6.13, and
   - E_m, E_{da}, E_{ua} as defined earlier in definition 6.5.

Definition 6.17: Let q_1 and q_2 be two class vertices in an ASG. A flow path from vertex q_1 to vertex q_2 is an association flow path, denoted as $q_1 \xrightarrow[ASF]{} q_2$, if and only if one of the following holds:

1. $\exists \delta$, $\delta \in V_m \cup V_a$, and $\exists \chi$, $\chi \in V_m$ such that $\delta \xrightarrow[E_{pub} \cup E_{pro} \cup E_{pri}]{} q_1 \wedge \delta \xrightarrow[Em]{} \chi \wedge \chi \xrightarrow[E_{pub} \cup E_{pro} \cup E_{pri}]{} q_2$, or

2. $\exists \alpha$, $\alpha \in V_c$, $\exists\, \delta'$, $\delta' \in V_m \cup V_a$, and $\exists \chi'$, $\chi' \in V_m$, such that $\delta' \xrightarrow[E_{pub} \cup E_{pro} \cup E_{pri}]{} q_1 \wedge \delta' \xrightarrow[Em]{} \chi' \wedge \chi' \xrightarrow[E_{pub} \cup E_{pro} \cup E_{pri}]{} \alpha$, and $\alpha \xrightarrow[ASF]{} q_2$.

The association graph and association flow paths of the example program are depicted in Figure 6.13 and 6.14.

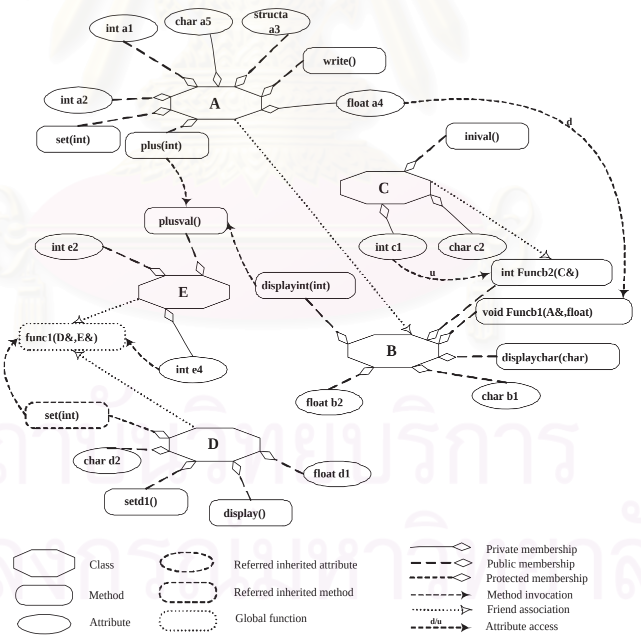


Figure 6.13: Association Graph

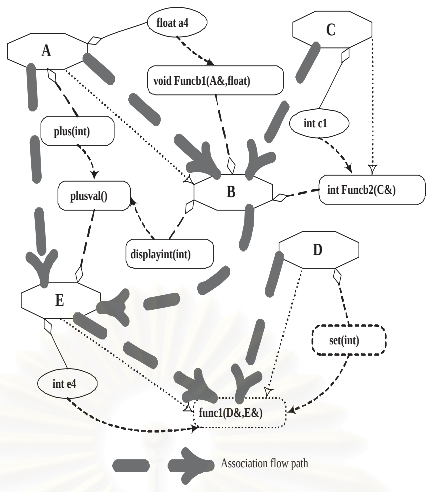


Figure 6.14: Association flow path

6.1.4 Control Flow

Control flow is a sequence of method invocation and related attributes which involve classes, methods, and attributes in the sequence.

Definition 6.18: Let P be an object-oriented program. A Control Flow Graph (CFG) of P is defined as $G_{CFG}(P) = (V, E)$, where (V, E) is a directed graph, and

1. $V = V_{sec} \cup V_c \cup V_m \cup V_a \cup V_{mri} \cup V_{ari}$ such that

    - V_{sec} is a set of vertices representing source code sections, and
    - V_c, V_m, V_a, V_{mri}, V_{ari} as defined earlier in definition 6.5.

2. $E = (E_{ctl}, E_{inv}, E_{pub}, E_{pro}, E_{pri}, E_m)$ such that

    - E_{ctl} is a set of edges from a source code section to another source code section,
    - E_{inv} is a set of edges from an involved class, attribute, or method to a source code section, and

Table 6.2: Object-Oriented program analysis graphs

Abbreviation	**Graph name**
PSG	Program Structure Graph
AAG	Attribute Access Graph
MIG	Method Invocation Graph
IFG	Inheritance Flow Graph
ASG	Association Graph
CFG	Control Flow Graph

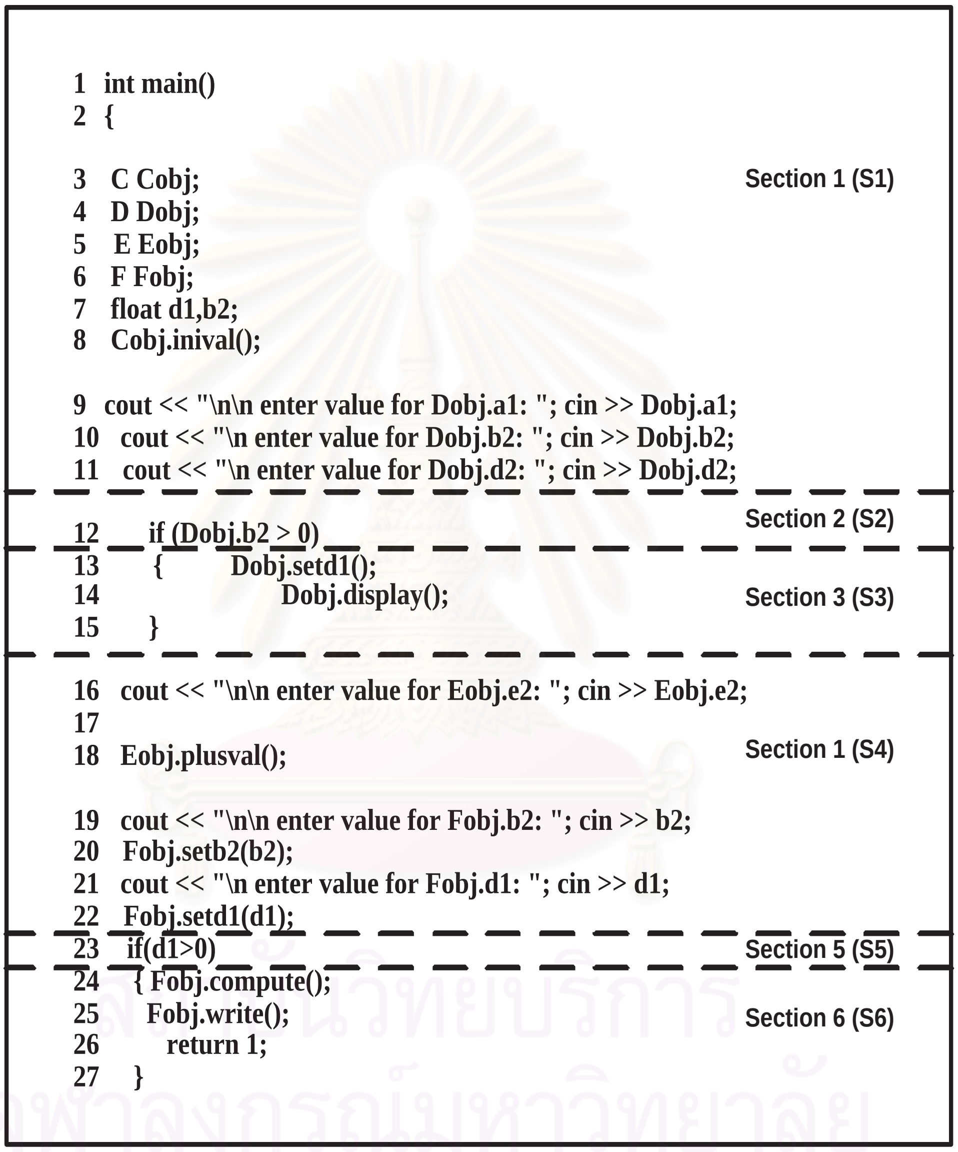
- E_{pub}, E_{pro}, E_{pri}, E_m as defined earlier in definition 6.5.

In a CFG, an edge from a class to a source code section (E_{inv}) means that an object of the class is only initiated without implementation as appeared on line 3 and 5-6 of the example program in Figure 6.15. Figure 6.16 shows a control flow graph of the program. Table 6.2 summarizes all the graphs so defined.

6.2 Static Fault and Dynamic Fault Analyses

In Mahaweerawat's, et al, previous work [4, 5], fault prediction was performed using software metrics obtained from source code measurement. The proposed MASP model acts as a fault metric selector that gathers relevant filtering metrics suitable for specific fault types. The coarse-grained algorithm selects important metrics for faultiness prediction while the fine-grained algorithm incorporates additional metrics which are essential for fault type identification. Since fault prediction was accomplished based on metrics gathered from source code, it was considered a static fault analysis.

However, there exists faults which occur at run time and may not be detected at static fault analysis stage. Consequently, dynamic fault analysis is employed to accommodate those faults by means of graphs. Each object-oriented program is represented by a set of graphs according to program structure, control flow, inheritance, association. The analyses start from coarse-grained to fine-grained stage to determine any dynamic faults



```
int main()
{

 C Cobj;
 D Dobj;
  E Eobj;
 F Fobj;
 float d1,b2;
 Cobj.inival();

cout << "\n\n enter value for Dobj.a1: "; cin >> Dobj.a1;
 cout << "\n enter value for Dobj.b2: "; cin >> Dobj.b2;
 cout << "\n enter value for Dobj.d2: "; cin >> Dobj.d2;

    if (Dobj.b2 > 0)
     {         Dobj.setd1();
                      Dobj.display();
    }

 cout << "\n\n enter value for Eobj.e2: "; cin >> Eobj.e2;

 Eobj.plusval();

 cout << "\n\n enter value for Fobj.b2: "; cin >> b2;
 Fobj.setb2(b2);
 cout << "\n enter value for Fobj.d1: "; cin >> d1;
 Fobj.setd1(d1);
  if(d1>0)
   { Fobj.compute();
      Fobj.write();
         return 1;
   }
```

Figure 6.15: Main function

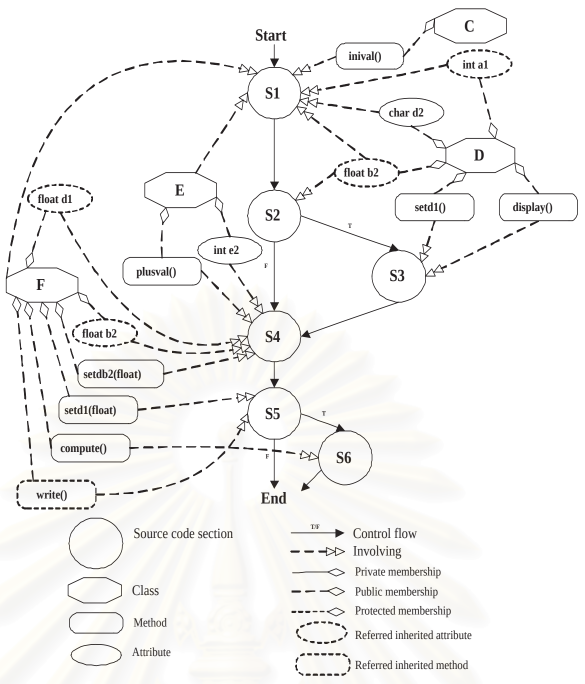


Figure 6.16: Control flow graph of main function

and the corresponding cause of fault that might exist. In the coarse-grained stage, control flow graph, attribute access graph, method invocation graph, and inheritance flow graph are employed to select relevant classes, attributes, and methods. The control flow graph is used to determine which classes and their members attribute to the control flow path in question, while the inheritance flow graph is used to extract additional classes that have inheritance relationship with the classes obtained from the control flow graph. In the fine-grained stage, the association graph are employed to explore other classes that might have association relationship, respectively, to the classes obtained from the control flow graph in the coarse-grained stage.

In addition to the classes and their members examined, all related fault metrics to those classes are also taken into account. The fault metrics are obtained from the implemented graphs and described in Section 6.3.

Table 6.3: Fault and anomalies due to inheritance

Acronym	Fault/Anomaly
SDA	State Definition Anomaly (possible post-condition violation)
SDIH	State Definition Inconsistency (due to state variable hiding)
SDI	State Definition Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
SVA	State Visibility Anomaly

6.3 Fault Metrics

This study focuses on five fault types incurred by the use of inheritance and polymorphism shown in Table 6.3. These metrics are established as a fault-tracer that will lead to locate the possible source of errors. Details on each fault type are described in [35]. A set of object-oriented software metrics and parameters [3,4,34,46] are applied to measure the source code under investigation. These metrics are summarized in Table 6.4, 6.5, 6.6, 6.7 and 6.8. The following sections elaborated the definition of each fault type.

Class

1. Program Structure: Let c be a class vertex in a PSG or AAG. The metrics of class c are described as follows:
   - Number of new methods (N_{newM}); N_{newM} is the number of methods $m \in V_m,\ m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
   - Number of new attributes (N_{newA}) N_{newA} is the number of attributes $a \in V_a,\ a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
   - Number of private attributes (N_{priA}); N_{priA} is the number of paths $\alpha \rightarrow c \in E_{pri}$ and $\alpha \in V_a \cup V_{ari}$

Table 6.4: Object-Oriented metrics for fault prediction

Metric	**Description**	**Domain of application**
N_{newM}	Number of new methods	Class/Program structure
N_{newA}	Number of new attributes	Class/Program structure
N_{priA}	Number of private attributes	Class/Program structure
N_{pubA}	Number of public attributes	Class/Program structure
N_{proA}	Number of protected attributes	Class/Program structure
N_{priM}	Number of private methods	Class/Program structure
N_{pubM}	Number of public methods	Class/Program structure
N_{proM}	Number of protected methods	Class/Program structure
N_{mdiav}	Number of new methods which define inherited attribute value	Class/Program structure
N_{muiav}	Number of new methods which use inherited attribute value	Class/Program structure
N_{ihp}	Number of inheritance paths	Class/Inheritance
N_{anc}	Number of ancestors	Class/Inheritance
N_{riaa}	Number of referred inherited attributes from ancestors	Class/Inheritance
N_{riap}	Number of referred inherited attributes from parents	Class/Inheritance
N_{ima}	Number of inherited methods which are originally defined by ancestors	Class/Inheritance
N_{imp}	Number of inherited methods which are originally defined by parents	Class/Inheritance
N_{rima}	Number of referred inherited methods from ancestors	Class/Inheritance
N_{rimp}	Number of referred inherited methods from parents	Class/Inheritance
N_{DSria}	Number of referred inherited attributes with signature-inheritance define	Class/Inheritance
N_{DSrim}	Number of referred inherited method with signature-inheritance define	Class/Inheritance
N_{DBrim}	Number of referred inherited methods with body-inheritance define	Class/Inheritance
N_{Urim}	Number of referred inherited methods with signature and body inheritance use	Class/Inheritance
$N_{iaDsbmin}$	Number of methods with signature and body inheritance define which are members of ancestors in the considered inheritance path and invoked by the considered class	Class/Inheritance
$N_{iaUsDbmin}$	Number of methods with signature inheritance use and body inheritance define which are members of ancestors in the considered inheritance path and invoked by the considered class	Class/Inheritance
$N_{iaUsbmin}$	Number of methods with signature and body inheritance use which are members of ancestors in the considered inheritance path and invoked by the considered class	Class/Inheritance
$N_{ipDsbmin}$	Number of methods with signature and body inheritance define which are members of parent in the considered inheritance path and invoked by the considered class	Class/Inheritance
$N_{ipUsDbmin}$	Number of methods with signature inheritance use and body inheritance define which are members of parent in the considered inheritance path and invoked by the considered class	Class/Inheritance
$N_{ipUsbmin}$	Number of methods with signature and body inheritance use which are members of parent in the considered inheritance path and invoked by the considered class	Class/Inheritance
N_{iaDsa}	Number of attributes with signature inheritance define which are members of ancestors in the considered inheritance path and accessed by the considered class	Class/Inheritance
N_{iaUsa}	Number of attributes with signature inheritance use which are members of ancestors in the considered inheritance path and accessed by the considered class	Class/Inheritance
N_{ipDsa}	Number of attributes with signature inheritance define which are members of parent in the considered inheritance path and accessed by the considered class	Class/Inheritance

Table 6.5: Object-Oriented metrics for fault prediction (continued)

Metric	**Description**	**Domain of application**
N_{ipUsa}	Number of attributes with signature inheritance use which are members of parent in the considered inheritance path and accessed by the considered class	Class/Inheritance
N_{par}	Number of parents	Class/Inheritance
$N_{riaatot}$	Number of referred inherited attributes from ancestors of all inheritance paths	Class/Inheritance
$N_{riaptot}$	Number of referred inherited attributes from parents of all inheritance paths	Class/Inheritance
$N_{rimatot}$	Number of referred inherited methods from ancestors of all inheritance paths	Class/Inheritance
$N_{rimptot}$	Number of referred inherited methods from parents of all inheritance paths	Class/Inheritance
N_{imatot}	Number of inherited methods from ancestors of all inheritance paths	Class/Inheritance
N_{imptot}	Number of inherited methods from parents of all inheritance paths	Class/Inheritance
$N_{iaDsbmintot}$	Number of methods with signature and body inheritance define which are members of ancestors and invoked by the considered class	Class/Inheritance
$N_{iaUsDbmintot}$	Number of methods with signature inheritance use and body inheritance define which are members of ancestors and invoked by the considered class	Class/Inheritance
$N_{iaUsbmintot}$	Number of methods with signature and body inheritance use which are members of ancestors and invoked by the considered class	Class/Inheritance
$N_{ipDsbmintot}$	Number of methods with signature and body inheritance define which are members of parents and invoked by the considered class	Class/Inheritance
$N_{ipUsDbmintot}$	Number of methods with signature inheritance use and body inheritance define which are members of parents and invoked by the considered class	Class/Inheritance
$N_{ipUsbmintot}$	Number of methods with signature and body inheritance use which are members of parents and invoked by the considered class	Class/Inheritance
$N_{iaDsatot}$	Number of attributes with signature inheritance define which are member of ancestors and accessed by the considered class	Class/Inheritance
$N_{iaUsatot}$	Number of attributes with signature inheritance use which are member of ancestors and accessed by the considered class	Class/Inheritance
$N_{ipDsatot}$	Number of attributes with signature inheritance define which are member of parents and accessed by the considered class	Class/Inheritance
$N_{ipUsatot}$	Number of attributes with signature inheritance use which are member of parents and accessed by the considered class	Class/Inheritance
DIT	Depth of inheritance	Class/Inheritance
$N_{DSriatot}$	Number of referred inherited attributes with signature-inheritance define from all inheritance paths	Class/Inheritance
$N_{DSrimtot}$	Number of referred inherited methods with signature-inheritance define from all inheritance paths	Class/Inheritance
$N_{DBrimtot}$	Number of referred inherited methods with body-inheritance define from all inheritanc paths	Class/Inheritance
$N_{Urimtot}$	Number of referred inherited methods with signature and body inheritance use from all inheritanc paths	Class/Inheritance
N_{ASOmin}	Number of methods which invoke other methods or access attributes in other classes with ordinary association	Class/Association
N_{ASFmin}	Number of methods which invoke other methods or access attributes in other classes with friend member function association	Class/Association
N_{ASCmin}	Number of methods which invoke other methods or access attributes in other classes with friend class association	Class/Association
$N_{ASOmout}$	Number of methods which are invoked by other methods in other classes with ordinary association	Class/Association
$N_{ASFmout}$	Number of methods which are invoked by other methods in other classes with friend member function association	Class/Association
$N_{ASCmout}$	Number of methods which are invoked by other methods in other classes with friend class association	Class/Association

Table 6.6: Object-Oriented metrics for fault prediction (continued)

Metric	**Description**	**Domain of application**
$N_{ASPmout}$	Number of methods which are invoked by global functions with friend operation association	Class/Association
N_{ASOa}	Number of attributes which are accessed by methods in other classes with ordinary association	Class/Association
N_{ASFa}	Number of attributes which are accessed by methods in other classes with friend member function association	Class/Association
N_{ASCa}	Number of attributes which are accessed by methods in other classes with friend class association	Class/Association
N_{ASPa}	Number of attributes which are accessed by global functions with friend operation association	Class/Association
N_{ASFcin}	Number of associated classes whose members are invoked/accessed by the considered class with friend class association	Class/Association
N_{ASFo}	Number of associated classes in friend operation association	Class/Association
$N_{ASFcmin}$	Number of associated classes whose members are invoked/accessed by the considered class with friend member function association	Class/Association
N_{ASOcin}	Number of associated classes whose members are invoked/accessed by the considered class with ordinary association	Class/Association
$N_{ASFcout}$	Number of associated classes whose members invoke/access members of the considered class with friend class association	Class/Association
$N_{ASFcmout}$	Number of associated classes whose members invoke/access members of the considered class with friend member function association	Class/Association
$N_{ASOcout}$	Number of associated classes whose members invoke/access members of the considered class with ordinary association	Class/Association
$AMtype$	Membership type of the considered attribute; private/protected/public	Attribute/Program structure
N_{atac}	Number of accesses via method invocation within the class	Attribute/Program structure
$N_{newMdef}$	Number of new methods which define the considered attribute value	Attribute/Program structure
$N_{newMuse}$	Number of new methods which use the considered attribute value	Attribute/Program structure
$N_{Diffmet}$	Number of inherited methods which implement the considered attribute in different way from the overridden method do	Attribute/Program structure
$SOperA$	Signature-operation on the consider attribute	Attribute/Inheritance
$A_{nw/ih}$	whether the considered attribute is newly defined or inherited from a superclass	Attribute/Inheritance
$N_{mDsbdef}$	Number of inherited methods with signature and body inheritance define which define the considered attribute value	Attribute/Inheritance
$N_{mUsDbdef}$	Number of inherited methods with signature inheritance use and body inheritance define which define the considered attribute value	Attribute/Inheritance
$N_{mUsbdef}$	Number of inherited methods with signature and body inheritance use which define the considered attribute value	Attribute/Inheritance
$N_{mDsbuse}$	Number of inherited methods with signature and body inheritance define which use the considered attribute value	Attribute/Inheritance
$N_{mUsDbuse}$	Number of inherited methods with signature inheritance use and body inheritance define which use the considered attribute value	Attribute/Inheritance
$N_{mUsbuse}$	Number of inherited methods with signature and body inheritance use which use the considered attribute value	Attribute/Inheritance
L_{odefa}	Level of class that originally defines the considered attribute	Attribute/Inheritance
N_{Aoras}	Number of accesses via ordinary association	Attribute/Association
N_{Afmas}	Number of accesses via friend member function association	Attribute/Association
N_{Afcas}	Number of accesses via friend class association	Attribute/Association
N_{Afoas}	Number of access via friend operation association	Attribute/Association
$MMtype$	Membership type of the considered method; private/protected/public	Method/Program structure

Table 6.7: Object-Oriented metrics for fault prediction (continued)

Metric	**Description**	**Domain of application**
N_{aacm}	Number of attributes accessed by the considered method within the same class	Method/Program structure
N_{mivin}	Number of other methods invoked by the considered method within the same class	Method/Program structure
$N_{newmivin}$	Number of other new methods invoked by the considered method with in the same class	Method/Program structure
N_{mivout}	Number of other methods which invoke the considered method within the same class	Method/Program structure
$N_{newmivout}$	Number of other new methods which invoke the considered method within the same class	Method/Program structure
$N_{diffatt}$	Number of inherited attributes which are implemented by the considered method in different way from the overridden method do	Method/Program structure
$SBOperM$	Signature-operation and Body-operation on the considered method; define/use	Method/Inheritance
$M_{nw/ih}$	whether the considered method is newly defined or inherited from a superclass	Method/Inheritance
N_{aDsdef}	Number of inherited attribute with signature inheritance define which their values are defined by the considered method	Method/Inheritance
N_{aUsdef}	Number of inherited attribute with signature inheritance use which their values are defined by the considered method	Method/Inheritance
N_{aDsuse}	Number of inherited attributes with signature inheritance define which their values are used by the considered method	Method/Inheritance
N_{aUsuse}	Number of inherited attributes with signature inheritance use which their values are used by the considered method	Method/Inheritance
$N_{Dsbmivin}$	Number of other inheritance methods with signature and body inheritance define which are invoked by the considered method with in the same class	Method/Inheritance
$N_{UsDbmivin}$	Number of other inheritance methods with signature inheritance use and body inheritance define which are invoked by the considered method with in the same class	Method/Inheritance
$N_{Usbmivin}$	Number of other inheritance methods with signature and body inheritance use which are invoked by the considered method with in the same class	Method/Inheritance
$N_{Dsbmivout}$	Number of other inheritance methods with signature and body inheritance define which invoke the considered method with in the same class	Method/Inheritance
$N_{UsDbmivout}$	Number of other inheritance methods with signature inheritance use and body inheritance define which invoke the considered method with in the same class	Method/Inheritance
$N_{Usbmivout}$	Number of other inheritance methods with signature and body inheritance use which invoke the considered method with in the same class	Method/Inheritance
$N_{actDsbmin}$	Number of methods with signature and body inheritance define which are members of ancestors and invoked by the considered method	Method/Inheritance
$N_{actUsDbmin}$	Number of methods with signature inheritance use and body inheritance define which are members of ancestors and invoked by the considered method	Method/Inheritance
$N_{actUsbmin}$	Number of methods with signature and body inheritance use which are members of ancestors and invoked by the considered method	Method/Inheritance
$N_{actDsatt}$	Number of attributes with signature inheritance define which are member of ancestors and accessed by the considered method	Method/Inheritance
$N_{actUsatt}$	Number of attributes with signature inheritance use which are member of ancestors and accessed by the considered method	Method/Inheritance
$N_{parDsbmin}$	Number of methods with signature and body inheritance define which are members of parents and invoked by the considered method	Method/Inheritance

Table 6.8: Object-Oriented metrics for fault prediction (continued)

Metric	**Description**	**Domain of application**
$N_{parUsDbmin}$	Number of methods with signature inheritance use and body inheritance define which are members of parents and invoked by the considered method	Method/Inheritance
$N_{parUsbmin}$	Number of methods with signature and body inheritance use which are members of parents and invoked by the considered method	Method/Inheritance
$N_{parDsatt}$	Number of attributes with signature inheritance define which are member of parents and accessed by the considered method	Method/Inheritance
$N_{parUsatt}$	Number of attributes with signature inheritance use which are member of parents and accessed by the considered method	Method/Inheritance
L_{odefm}	Level of class that originally defines the considered method	Method/Inheritance
N_{Moriv}	Number of invocations via ordinary association	Method/Association
N_{Mfmiv}	Number of invocations via friend member function association	Method/Association
N_{Mfciv}	Number of invocations via friend class association	Method/Association
N_{Mfoiv}	Number of invocations via friend operation association	Method/Association
N_{Masoin}	Number of other methods invoked by the considered method via ordinary association	Method/Association
N_{Masfin}	Number of other methods invoked by the considered method via friend member function association	Method/Association
N_{Mascin}	Number of other methods invoked by the considered method via friend class association	Method/Association

- Number of public attributes (N_{pubA}); N_{pubA} is the number of paths $\alpha \rightarrow c \in E_{pub}$ and $\alpha \in V_a \cup V_{ari}$
- Number of protected attributes (N_{proA}); N_{proA} is the number of paths $\alpha \rightarrow c \in E_{pro}$ and $\alpha \in V_a \cup V_{ari}$
- Number of private methods (N_{priM}); N_{priM} is the number of paths $\alpha \rightarrow c \in E_{pri}$ and $\alpha \in V_m \cup V_{mri}$
- Number of public methods (N_{pubM}); N_{pubM} is the number of paths $\alpha \rightarrow c \in E_{pub}$ and $\alpha \in V_m \cup V_{mri}$
- Number of protected methods (N_{proM}); N_{proM} is the number of paths $\alpha \rightarrow c \in E_{pro}$ and $\alpha \in V_m \cup V_{mri}$
- Number of new methods which define inherited attribute values (N_{mdiav}); N_{mdiav} is the number of methods $m \in V_m$, $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and

$\exists\, \delta \in V_{ari},\ \delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and the following conditions are true:

(i) $\delta \rightarrow m \in E_{da}$, or

(ii) $\exists\, p,\ p \in V_{mri} \cup V_m$ and $p \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $p \rightarrow m \in E_m$ and $\delta \rightarrow p \in E_{da}$

- Number of new methods which use inherited attribute values (N_{muiav}); N_{muiav} is the number of methods $m \in V_m,\ m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_{ari},\ \delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and the following conditions are true:

  (i) $\delta \rightarrow m \in E_{ua}$, or

  (ii) $\exists\, p,\ p \in V_{mri} \cup V_m$ and $p \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $p \rightarrow m \in E_m$ and $\delta \rightarrow p \in E_{ua}$

2. Inheritance: Let c be a class vertex in an IFG. The metrics of class c are described as follows:

   - Number of inheritance paths (N_{ihp}); N_{ihp} is the number of class vertices $\delta \in V_c$ and there exists a path $\delta \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$ and for all $\alpha \in V_c$, $\alpha \rightarrow \delta \in E_{hpub} \cup E_{hpro} \cup E_{hpri} \ = \ \varnothing$
   - For each inheritance path:
     - Number of ancestors (N_{anc}); N_{anc} is the number of class vertices $\alpha \in V_c$ which have a path from itself to the considered class c; $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
     - Number of referred inherited attributes from ancestors (N_{riaa}); N_{riaa} is the number of referred inherited attribute vertices $a \in V_{ari}$ and $a \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $p \in V_a \cup V_{ari}, p \rightarrow a \in E_{mh}$ and $\alpha \in V_c, p \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpup} \cup E_{hpro} \cup E_{hpri}$

- Number of referred inherited attributes from parents (N_{riap}); N_{riap} is the number of referred inherited attribute vertices $a \in V_{ari}$ and $a \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $p \in V_a \cup V_{ari}, p \rightarrow a \in E_{mh}$ and $\alpha \in V_c, p \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\, \alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of inherited methods which are originally defined by ancestors (N_{ima}); N_{ima} is the number of method vertices $m \in V_m$ and $\alpha \in V_c$, $m \rightarrow \alpha \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of inherited methods which are originally defined by parents (N_{imp}); N_{imp} is the number of method vertices $m \in V_m$ and $\alpha \in V_c$, $m \rightarrow \alpha \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $\exists\, \alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of referred inherited methods from ancestors (N_{rima}); N_{rima} is the number of referred inherited method vertices $m \in V_{mri}$ and $m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $k \in V_m, k \rightarrow m \in E_{mh}$ and $\alpha \in V_c, k \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpup} \cup E_{hpro} \cup E_{hpri}$
- Number of referred inherited methods from parents (N_{rimp}); N_{rimp} is the number of referred inherited method vertices $m \in V_{mri}$ and $m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $k \in V_m, k \rightarrow m \in E_{mh}$ and $\alpha \in V_c, k \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\, \alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of referred inherited attributes with signature-inheritance define (N_{DSria}); N_{DSria} is the number of referred inherited attribute vertices $a \in V_{ari}$ and $a \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $T(TV(c \rightarrow a)) = \langle D_{si},\ N_{bi} \rangle$
- Number of referred inherited methods with signature-inheritance define (N_{DSrim}); N_{DSrim} is the number of referred inherited method vertices $m \in V_{mri}$ and $m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $T(TV(c \rightarrow m)) = \langle D_{si},\ D_{bi} \rangle$

– Number of referred inherited methods with body-inheritance define (N_{DBrim}); N_{DBrim} is the number of referred inherited method vertices $m \in V_{mri}$ and $m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $T(TV(c \rightarrow m)) \in \{\langle D_{si}, D_{bi}\rangle, \langle U_{si}, D_{bi}\rangle\}$

– Number of referred inherited methods with signature-inheritance use and body-inheritance use (N_{Urim}); N_{Urim} is the number of referred inherited method vertices $m \in V_{mri}$ and $m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $T(TV(c \rightarrow m)) = \langle U_{si}, U_{bi}\rangle$

– Number of methods with signature and body inheritance define ($\langle D_s, D_b\rangle$) which are members of ancestors in the considered inheritance path and invoked by the considered class ($N_{iaDsbmin}$); $N_{iaDsbmin}$ is the number of methods m, $m \in V_m \cup V_{mri}$, T(m) $= \langle D_s, D_b\rangle$, $\exists\, \alpha \in V_c$, $m \rightarrow \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $m \rightarrow \delta \in E_m$ and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$

– Number of methods with signature-inheritance use and body inheritance define ($\langle U_s, D_b\rangle$) which are members of ancestors in the considered inheritance path and invoked by the considered class ($N_{iaUsDbmin}$); $N_{iaUsDbmin}$ is the number of methods $m, m \in V_m \cup V_{mri}$, T(m) $= \langle U_s, D_b\rangle$, $\exists \alpha \in V_c, m \rightarrow \alpha \in E_{pro} \cup E_{pub}$ and $\exists \delta \in V_m \cup V_{mri}$, $\delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $m \rightarrow \delta \in E_m$ and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$

– Number of methods with signature and body inheritance use ($\langle U_s, U_b\rangle$) which are members of ancestors in the considered inheritance path and invoked by the considered class ($N_{iaUsbmin}$); $N_{iaUsbmin}$ is the number of methods m, $m \in V_m \cup V_{mri}$, T(m) $= \langle U_s, U_b\rangle$, $\exists\, \alpha \in V_c$, $m \rightarrow \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $m \rightarrow \delta \in E_m$

and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$

- Number of methods with signature and body inheritance define ($\langle D_s, D_b \rangle$) which are members of parent in the considered inheritance path and invoked by the considered class ($N_{ipDsbmin}$); $N_{ipDsbmin}$ is the number of methods m, $m \in V_m \cup V_{mri}$, $\mathrm{T(m)} = \langle D_s, D_b \rangle$, $\exists\, \alpha \in V_c$, $m \to \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $m \to \delta \in E_m$ and $\alpha \to c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of methods with signature-inheritance use and body-inheritance define ($\langle U_s, D_b \rangle$) which are members of parent in the considered inheritance path and invoked by the considered class ($N_{ipUsDbmin}$); $N_{ipUsDbmin}$ is the number of methods m, $m \in V_m \cup V_{mri}$, $\mathrm{T(m)} = \langle D_s, D_b \rangle$, $\exists\, \alpha \in V_c$, $m \to \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $m \to \delta \in E_m$ and $\alpha \to c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of methods with signature and body inheritance use ($\langle U_s, U_b \rangle$) which are members of parent in the considered inheritance path and invoked by the considered class ($N_{ipUsbmin}$); $N_{ipUsbmin}$ is the number of methods m, $m \in V_m \cup V_{mri}$, $\mathrm{T(m)} = \langle D_s, D_b \rangle$, $\exists\, \alpha \in V_c$, $m \to \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $m \to \delta \in E_m$ and $\alpha \to c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of attributes with signature-inheritance define ($\langle D_s, N_b \rangle$) which are members of ancestors in the considered inheritance path and accessed by the considered class (N_{iaDsa}); N_{iaDsa} is the number of attributes a, $a \in V_a \cup V_{ari}$, $\mathrm{T(a)} = \langle D_s, N_b \rangle$, $\exists\, \alpha \in V_c$, $a \to \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $a \to \delta \in E_{da} \cup E_{ua}$ and $\exists \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$

  - Number of attributes with signature-inheritance use ($\langle U_s, N_b\rangle$) which are members of ancestors in the considered inheritance path and accessed by the considered class (N_{iaUsa}); N_{iaUsa} is the number of attributes a, $a \in V_a \cup V_{ari}$, T(a) $= \langle D_s, N_b\rangle$, $\exists\, \alpha \in V_c$, $a \rightarrow \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $a \rightarrow \delta \in E_{da} \cup E_{ua}$ and $\exists \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$
  - Number of attributes with signature-inheritance define ($\langle D_s, N_b\rangle$) which are members of parent in the considered inheritance path and accessed by the considered class (N_{ipDsa}); N_{ipDsa} is the number of attributes a, $a \in V_a \cup V_{ari}$, T(a) $= \langle D_s, N_b\rangle$, $\exists\, \alpha \in V_c$, $a \rightarrow \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $a \rightarrow \delta \in E_{da} \cup E_{ua}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
  - Number of attributes with signature-inheritance use ($\langle U_s, N_b\rangle$) which are members of parent in the considered inheritance path and accessed by the considered class (N_{ipUsa}); N_{ipUsa} is the number of attributes a, $a \in V_a \cup V_{ari}$, T(a) $= \langle U_s, N_b\rangle$, $\exists\, \alpha \in V_c$, $a \rightarrow \alpha \in E_{pro} \cup E_{pub}$ and $\exists\, \delta \in V_m \cup V_{mri}$, $\delta \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, $a \rightarrow \delta \in E_{da} \cup E_{ua}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$

- Number of parents (N_{par}); N_{par} is the number of class vertices $\alpha \in V_c$ and $\exists\, \alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of referred inherited attributes from ancestors of all inheritance paths ($N_{riaatot}$); $N_{riaatot} = \sum_{j=1}^{N_{ihp}} N_{riaa}^{j}$, N_{riaa}^{j} is number of referred inherited attributes from ancestors of the inheritance path j, $j = \{1,\, 2, \ldots,\, N_{ihp}\}$
- Number of referred inherited attributes from parents of all inheritance paths ($N_{riaptot}$); $N_{riaptot} = \sum_{j=1}^{N_{ihp}} N_{riap}^{j}$, N_{riap}^{j} is number of referred inherited attributes from parent of the inheritance path j, $j = \{1,\, 2, \ldots,\, N_{ihp}\}$

- Number of referred inherited methods from ancestors of all inheritance paths ($N_{rimatot}$); $N_{rimatot} = \sum_{j=1}^{N_{ihp}} N_{rima}^{j}$, N_{rima}^{j} is number of referred inherited methods from ancestors of the inheritance path j, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of referred inherited methods from parents of all inheritance paths ($N_{rimptot}$); $N_{rimptot} = \sum_{j=1}^{N_{ihp}} N_{rimp}^{j}$, N_{rimp}^{j} is number of referred inherited methods from parents of the inheritance path j, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of inherited methods which are originally defined by ancestors from all inheritance paths (N_{imatot}); $N_{imatot} = \sum_{j=1}^{N_{ihp}} N_{ima}^{j}$, N_{ima}^{j} is number of inherited methods from ancestors of the inheritance path j, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of inherited methods which are originally defined by parents from all inheritance paths (N_{imptot}); $N_{imptot} = \sum_{j=1}^{N_{ihp}} N_{imp}^{j}$, N_{imp}^{j} is number of inherited methods from parent of the inheritance path j, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of methods with signature and body inheritance define ($\langle D_s, D_b \rangle$) which are members of ancestors and invoked by the considered class ($N_{iaDsbmintot}$); $N_{iaDsbmintot} = \sum_{j=1}^{N_{ihp}} N_{iaDsbmin}^{j}$, $N_{iaDsbmin}^{j}$ is the number of methods with $\langle D_s, D_b \rangle$ which are members of ancestors in the inheritance path j and invoked by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of methods with signature inheritance use and body inheritance define ($\langle U_s, D_b \rangle$) which are members of ancestors and invoked by the considered class ($N_{iaUsDbmintot}$); $N_{iaUsDbmintot} = \sum_{j=1}^{N_{ihp}} N_{iaUsDbmin}^{j}$, $N_{iaUsDbmin}^{j}$ is the number of methods with $\langle U_s, D_b \rangle$ which are members of ancestors in the inheritance path j and invoked by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of methods with signature and body inheritance use ($\langle U_s, U_b \rangle$) which are members of ancestors and invoked by the considered class ($N_{iaUsbmintot}$); $N_{iaUsbmintot} = \sum_{j=1}^{N_{ihp}} N_{iaUsbmin}^{j}$, $N_{iaUsbmin}^{j}$ is the number of methods with

$\langle U_s, U_b\rangle$ which are members of ancestors in the inheritance path j and invoked by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$

- Number of methods with signature and body inheritance define ($\langle D_s, D_b\rangle$) which are members of parents and invoked by the considered class ($N_{ipDsbmintot}$); $N_{ipDsbmintot} = \sum_{j=1}^{N_{ihp}} N^j_{ipDsbmin}$, $N^j_{ihDsbmin}$ is the number of methods which are members of parent in the inheritance path j and invoked by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of methods with signature inheritance use and body inheritance define ($\langle U_s, D_b\rangle$) which are members of parents and invoked by the considered class ($N_{ipUsDbmintot}$); $N_{ipUsDbmintot} = \sum_{j=1}^{N_{ihp}} N^j_{ipUsDbmin}$, $N^j_{ihUsDbmin}$ is the number of methods which are members of parents in the inheritance path j and invoked by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of methods with signature and body inheritance use ($\langle U_s, U_b\rangle$) which are members of parents and invoked by the considered class ($N_{ipUsbmintot}$); $N_{ipUsbmintot} = \sum_{j=1}^{N_{ihp}} N^j_{ipUsbmin}$, $N^j_{ihUsbmin}$ is the number of methods which are members of parent in the inheritance path j and invoked by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of attributes with signature-inheritance define ($\langle D_s, N_b\rangle$) which are members of ancestors and accessed by the considered class ($N_{iaDsatot}$); $N_{iaDsatot} = \sum_{j=1}^{N_{ihp}} N^j_{iaDsa}$, N^j_{iaDsa} is the number of attributes with $\langle D_s, N_b\rangle$ which are members of ancestors in the inheritance path j and accessed by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$
- Number of attributes with signature-inheritance use ($\langle U_s, N_b\rangle$) which are members of ancestors and accessed by the considered class ($N_{iaUsatot}$); $N_{iaUsatot} = \sum_{j=1}^{N_{ihp}} N^j_{iaUsa}$, N^j_{iaUsa} is the number of attributes with $\langle U_s, N_b\rangle$ which are

members of ancestors in the inheritance path j and accessed by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$

- Number of attributes with signature-inheritance use ($\langle U_s, N_b \rangle$) which are members of parents and accessed by the considered class ($N_{ipUsatot}$); $N_{ipUsatot} = \sum_{j=1}^{N_{ihp}} N^j_{ipUsa}$, N^j_{ipUsa} is the number of attributes with $\langle U_s, N_b \rangle$ which are members of parent in the inheritance path j and accessed by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$

- Number of attributes with signature-inheritance define ($\langle D_s, N_b \rangle$) which are members of parents and accessed by the considered class ($N_{ipDsatot}$); $N_{ipDsatot} = \sum_{j=1}^{N_{ihp}} N^j_{ipDsa}$, N^j_{ipDsa} is the number of attributes with $\langle D_s, N_b \rangle$ which are members of parent in the inheritance path j and accessed by the considered class, $j = \{1, 2, \ldots, N_{ihp}\}$

- Depth of inheritance (DIT); DIT $= \max_j N^j_{anc}$, where N^j_{anc} is number of ancestors of inheritance path j, $j = \{1, 2, \ldots, N_{ihp}\}$

- Number of referred inherited attributes with signature-inheritance define from all inheritance paths ($N_{DSriatot}$); $N_{DSriatot} = \left| \bigcup_{j=1}^{Nihp} A^j \right|$, A^j is a set of referred inherited attributes with signature-inheritance define from all inheritance paths j, $j = \{1, 2, \ldots, N_{ihp}\}$

- Number of referred inherited methods with signature-inheritance define from all inheritance paths ($N_{DSrimtot}$); $N_{DSrimtot} = \left| \bigcup_{j=1}^{Nihp} M^j_s \right|$, M^j_s is a set of referred inherited methods with signature-inheritance define from all inheritance paths j, $j = \{1, 2, \ldots, N_{ihp}\}$

- Number of referred inherited methods with body-inheritance define from all inheritance paths ($N_{DBrimtot}$); $N_{DBrimtot} = \left| \bigcup_{j=1}^{Nihp} M^j_b \right|$, M^j_b is a set of referred inherited methods with body-inheritance define from all inheritance paths j,

$j = \{1, 2, \ldots, N_{ihp}\}$

- Number of referred inherited methods with signature-inheritance use and body-inheritance use from all inheritance paths ($N_{Urimtot}$);
  $N_{Urimtot} = \left|\bigcup_{j=1}^{Nihp} M_u^j\right|$, M_u^j is a set of referred inherited methods with signature-inheritance use and body-inheritance use from all inheritance paths j, $j = \{1, 2, \ldots, N_{ihp}\}$

3. Association: Let c be a class vertex in an ASG. The metrics of class c are described as follows:

   - Number of methods which invoke other methods or access attributes in other classes with ordinary association (N_{ASOmin}); N_{ASOmin} is the number of methods or referred inherited method vertices $m \in V_m \cup V_{mri}, m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $\exists\, k \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ k \rightarrow m \in E_m \cup E_{da} \cup E_{ua}$ and $\forall\, \alpha \in V_c$ and $k \rightarrow \alpha \in E_{pub}$ and $\alpha \xrightarrow[E_{fa}]{} c = \varnothing$ and $\alpha \xrightarrow[E_{fa}]{} m = \varnothing$
   - Number of methods which invoke other methods or access attributes in other classes with friend member function association (N_{ASFmin}); N_{ASFmin} is the number of methods or referred inherited method vertices $m \in V_m \cup V_{mri}, m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $\exists k \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ k \rightarrow m \in E_m \cup E_{da} \cup E_{ua}$ and $\forall\, \alpha \in V_c$, $k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\alpha \rightarrow m \in E_{fa}$
   - Number of methods which invoke other methods or access attributes in other classes with friend class association (N_{ASCmin}); N_{ASCmin} is the number of method vertices $m \in V_m \cup V_{mri},\ m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $\exists\, k \in V_m \cup V_{mri} \cup V_a \cup V_{ari}, k \rightarrow m \in E_m \cup E_{da} \cup E_{ua}$ and $\forall \alpha \in V_c$, $k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ $k \rightarrow \delta \in E_{pub}$ and $\alpha \rightarrow c \in E_{fa}$
   - Number of methods which are invoked by other methods in other classes with ordinary association ($N_{ASOmout}$); $N_{ASOmout}$ is the number of method vertices

$m \in V_m \cup V_{mri}$, $m \to c \in E_{pub}$ and $\exists\, k \in V_m \cup V_{mri}$, $m \to k \in E_m$ and $\forall\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \xrightarrow[E_{fa}]{} \alpha = \varnothing$ and $c \xrightarrow[E_{fa}]{} k = \varnothing$

- Number of methods which are invoked by other methods in other classes with friend member function association ($N_{ASFmout}$); $N_{ASFmout}$ is the number of method vertices $m \in V_m \cup V_{mri}$, $m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, k \in V_m \cup V_{mri}$, $m \to k \in E_m$ and $\exists\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \to k \in E_{fa}$
- Number of methods which are invoked by other methods in other classes with friend class association ($N_{ASCmout}$); $N_{ASCmout}$ is the number of method vertices $m \in V_m \cup V_{mri}$, $m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, k \in V_m \cup V_{mri}$, $m \to k \in E_m$ and $\forall\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \to \alpha \in E_{fa}$
- Number of methods which are invoked by global functions with friend operation association ($N_{ASPmout}$); $N_{ASPmout}$ is the number of methods $m \in V_m \cup V_{mri}$, $m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, k \in V_{gf}$, $m \to k \in E_m$ and $c \to k \in E_{fa}$
- Number of attributes which are accessed by methods in other classes with ordinary association (N_{ASOa}); N_{ASOa} is the number of attribute vertices $a \in V_a \cup V_{ari}$, $a \to c \in E_{pub}$ and $\exists\, k \in V_m \cup V_{mri}$, $a \to k \in E_{da} \cup E_{ua}$ and $\forall\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \xrightarrow[E_{fa}]{} \alpha = \varnothing$ and $c \xrightarrow[E_{fa}]{} k = \varnothing$
- Number of attributes which are accessed by methods in other classes with friend member function association (N_{ASFa}); N_{ASFa} is the number of attribute vertices $a \in V_a \cup V_{ari}$, $a \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, k \in V_m \cup V_{mri}$, $a \to k \in E_{da} \cup E_{ua}$ and $\forall\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \to k \in E_{fa}$
- Number of attributes which are accessed by methods in other classes with friend class association (N_{ASCa}); N_{ASCa} is the number of attribute vertices

$a \in V_a \cup V_{ari}$, $a \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists k \in V_m \cup V_{mri}$, $a \to k \in E_{da} \cup E_{ua}$ and $\forall\, \alpha \in V_c,\ k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \to \alpha \in E_{fa}$

- Number of attributes which are accessed by global functions with friend operation association (N_{ASPa}); N_{ASPa} is the number of attribute vertices $a \in V_a \cup V_{ari},\ a \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, k \in V_{gf},\ a \to k \in E_{da} \cup E_{ua}$ and $c \to k \in E_{fa}$

- Number of associated classes whose members are invoked/accessed by the considered class c with friend class association (N_{ASFcin}); N_{ASFcin} is the number of class vertices $\alpha \in V_c$ and $\exists\, k \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ \ k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, m \in V_m \cup V_{mri},\ m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $k \to m \in E_m$ and $\alpha \to c \in E_{fa}$

- Number of associated classes in friend operation association (N_{ASFo}); N_{ASFo} is the number of class vertices $\alpha \in V_c$, there exists a global function $f \in V_{gf}$ and $\alpha \to f \in E_{fa}$ and $c \to f \in E_{fa}$

- Number of associated classes whose members are invoked/accessed by the considered class c with friend member function association ($N_{ASFcmin}$); $N_{ASFcmin}$ is the number of class vertices $\alpha \in V_c$, $\exists\, k \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, m \in V_m \cup V_{mri},\ m \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $k \to m \in E_m \cup E_{da} \cup E_{ua}$ and $\alpha \to m \in E_{fa}$

- Number of associated classes whose members are invoked/accessed by the considered class c with ordinary association (N_{ASOcin}); N_{ASOcin} is the number of class vertices $\alpha \in V_c,\ \exists\, k \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ k \to \alpha \in E_{pub}$ and $m \in V_m \cup V_{mri}, m \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $k \to m \in E_m \cup E_{da} \cup E_{ua}$ and $\alpha \underset{E_{fa}}{\longrightarrow} c = \varnothing$ and $\alpha \underset{E_{fa}}{\longrightarrow} m = \varnothing$

- Number of associated classes whose members invoke/access members of the considered class c with friend class association ($N_{ASFcout}$); $N_{ASFcout}$ is the number of class vertices $\alpha \in V_c$ and $\exists k \in V_m \cup V_{mri}, k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, m \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \to k \in E_m \cup E_{da} \cup E_{ua}$ and $c \to \alpha \in E_{fa}$
- Number of associated classes whose members invoke/access members of the considered class c with friend member function association ($N_{ASFcmout}$); $N_{ASFcmout}$ is the number of class vertices $\alpha \in V_c$, $\exists\, k \in V_m \cup V_{mri},\ k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, m \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ m \to c \in E_{pub} \cup E_{pro} \cup E_{pri}$ and $m \to k \in E_m \cup E_{da} \cup E_{ua}$ and $c \to k \in E_{fa}$
- Number of associated classes whose members invoke/access members of the considered class c with ordinary association ($N_{ASOcout}$); $N_{ASOcout}$ is the number of class vertices $\alpha \in V_c$, $\exists\, k \in V_m \cup V_{mri},\ k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \in V_m \cup V_{mri} \cup V_a \cup V_{ari},\ m \to c \in E_{pub}$ and $m \to k \in E_m \cup E_{da} \cup E_{ua}$ and $c \xrightarrow[E_{fa}]{} \alpha = \varnothing$ and $c \xrightarrow[E_{fa}]{} k = \varnothing$

Attribute

1. Program Structure, attribute access, and method invocation: Let a be an attribute vertex or a referred inherited attribute vertex and c be a class vertex in a PSG, an AAG, or an MIG. The metrics of attribute a are described as follows:

   - Membership Type; private/public/protected ($AMtype$): Determine whether membership relation of the consider attribute is private, public, or protected. An attribute a is said to be a private member of class c if and only if there exists an edge from the class vertex c to attribute vertex a; $\exists\, c \to a$ and $c \to a\ \in E_{pri}$. Public and protected membership are defined similarly using

E_{pub} and E_{pro}, respectively.

- Number of accesses via method invocation within the class (N_{atac}): N_{atac} is the number of edges from attribute a to method m; $a \rightarrow m \in E_{da} \cup E_{ua}$
- Number of new methods which define the considered attribute value ($N_{newMdef}$); $N_{newMdef}$ is the number of methods $m \in V_m, a \rightarrow m \in E_{da}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub},\ m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of new methods which use the considered attribute value ($N_{newMuse}$); $N_{newMuse}$ is the number of method $m \in V_m, a \rightarrow m \in E_{ua}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub},\ m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of inherited methods which implement the considered attribute in different way from the overridden methods do ($N_{Diffmet}$);$N_{Diffmet}$ is the number of inherited methods which access the considered attribute or call other methods accessing the considered attribute to yield a net effect on the considered attribute not equivalent to that of the inherited attribute of the ancestors made by the overridden methods [35].

2. Inheritance: Let a be an attribute vertex or a referred inherited attribute vertex in an IFG, MIG, and AAG. The metrics of attribute a are described as follows:

   - Signature-operation ($SOperA$); define/use: Determine the vertex tag of the considered attribute vertex a whether $T(a)$ is $\langle D_{si},\ N_{bi}\rangle$ or $\langle U_{si},\ N_{bi}\rangle$
   - new/inherited ($A_{nw/ih}$); whether it is newly defined or inherited from a superclass. An attribute a is newly defined if $T(a) = \langle D_{si},\ N_{bi}\rangle$. Otherwise the attribute a is inherited from a superclass, i.e., $T(a) = \langle U_{si},\ N_{bi}\rangle$
   - Number of inherited methods with signature and body inheritance define ($\langle D_s, D_b\rangle$) which define the considered attribute value ($N_{mDsbdef}$); $N_{mDsbdef}$

is the number of inherited methods $m \in V_{mri}, T(m) = \langle D_s, D_b \rangle, a \rightarrow m \in E_{da}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$

- Number of inherited methods with signature inheritance use and body inheritance define ($\langle U_s, D_b \rangle$) which define the considered attribute value ($N_{mUsDbdef}$); $N_{mUsDbdef}$ is the number of inherited methods $m \in V_{mri}, T(m) = \langle U_s, D_b \rangle, a \rightarrow m \in E_{da}$ and $\exists c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of inherited methods with signature and body inheritance use ($\langle U_s, U_b \rangle$) which define the considered attribute value ($N_{mUsbdef}$); $N_{mUsbdef}$ is the number of inherited methods $m \in V_{mri},\ T(m) = \langle U_s, U_b \rangle,\ a \rightarrow m \in E_{da}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of inherited methods with signature and body inheritance define ($\langle D_s, D_b \rangle$) which use the considered attribute value ($N_{mDsbuse}$); $N_{mDsbuse}$ is the number of inherited methods $m \in V_{mri},\ T(m) = \langle D_s, D_b \rangle,\ a \rightarrow m \in E_{ua}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of inherited methods with signature inheritance use and body inheritance use ($\langle U_s, D_b \rangle$) which define the considered attribute value ($N_{mUsDbuse}$); $N_{mUsDbuse}$ is the number of inherited methods $m \in V_{mri}, T(m) = \langle U_s, D_b \rangle, a \rightarrow m \in E_{ua}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of inherited methods with signature and body inheritance use ($\langle U_s, U_b \rangle$) which define the considered attribute value ($N_{mUsbuse}$); $N_{mUsbuse}$ is the number of inherited methods $m \in V_{mri},\ T(m) = \langle U_s, U_b \rangle,\ a \rightarrow m \in E_{ua}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Level of class that originally defines the considered attribute (L_{odefa}): Let c_1 and c_2 be class vertices in an IFG; $\{c_1,\ c_2\} \subseteq V_c$. Let the considered attribute

a be a member of class c_1, $a \to c_1 \in E_{pri} \cup E_{pub} \cup E_{pro}$. c_2 is said to be the class that originally defines attribute a if and only if the following conditions are true:

(i) $a \to c_2 \in E_{pub} \cup E_{pro}$ and $T(TV(c_2 \to a)) = \langle D_{si}, N_{bi} \rangle$

(ii) $T(TV(c_1 \to a)) = \langle U_{si}, N_{bi} \rangle$

(iii) $c_2 \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} c_1$ and

(iv) for all α, $\alpha \in V_c \wedge c_2 \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} \alpha \wedge \alpha \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} c_1$, such that $T(TV(\alpha \to a)) = \langle U_{si}, N_{bi} \rangle$

L_{odefa} is the number of classes along the path $c_2 \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} c_1$ which includes c_2 but excludes c_1.

3. Association: Let a be an attribute vertex or a referred inherited attribute vertex and c be a class vertex in an ASG. The attribute a is a member of class c, $a \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$. The metrics of attribute a are described as follows:

- Number of accesses via ordinary association (N_{Aoras}): N_{Aoras} is the number of edges from a to k; $a \to k \in E_{da} \cup E_{ua}$, $k \in V_m \cup V_{mri}$ and $a \to c \in E_{pub}$ and $\forall\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \xrightarrow[E_{fa}]{} \alpha = \varnothing$ and $c \xrightarrow[E_{fa}]{} k = \varnothing$

- Number of accesses via friend member function association (N_{Afmas}): N_{Afmas} is the number of edges from a to k; $a \to k \in E_{da} \cup E_{ua}$, $k \in V_m \cup V_{mri}$ and $a \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \to k \in E_{fa}$

- Number of accesses via friend class association (N_{Afcas}): N_{Afcas} is the number of edges from a to k; $a \to k \in E_{da} \cup E_{ua}$, $k \in V_m \cup V_{mri}$ and $a \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \alpha \in V_c$, $k \to \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \to \alpha \in E_{fa}$

- Number of accesses via friend operation association (N_{Afoas}): N_{Afoas} is the number of edges from a to k; $a \rightarrow k \in E_{da} \cup E_{ua}$, $k \in V_{gf}$ and $a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \rightarrow k \in E_{fa}$

Method

1. Program Structure, attribute access, and method invocation: Let m be a method vertex or a referred inherited method vertex and c be a class vertex in a PSG, an AAG, or an MIG. The metrics of method m are described as follows:

    - Membership Type; private/public/protected ($MMtype$): Determine whether membership relation of the considered method is private, public, or protected. A method m is said to be a private member of class c if and only if there exists an edge from the class vertex c to method vertex m; $\exists\, c \rightarrow m$ and $c \rightarrow m \in E_{pri}$, N_{atac} is the number of edges from attribute a to method m; $a \rightarrow m \in E_m$. Public and protected membership type are defined similarly using E_{pub} and E_{pro}, respectively.
    - Number of attributes accessed by the considered method within the same class (N_{aacm}); N_{aacm} is the number of edges from attribute $a \in V_a \cup V_{ari}$, $a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ to the considered method $m \in V_m \cup V_{mri}$, $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$; $a \rightarrow m \in E_{da} \cup E_{ua}$
    - Number of other methods invoked by the considered method within the same class (N_{mivin}); N_{mivin} is the number of edges from a method $k \in V_m \cup V_{mri}$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ to the considered method $m \in V_m \cup V_{mri}$, $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$; $k \rightarrow m \in E_m$
    - Number of other new methods invoked by the considered method within the same class ($N_{newmivin}$); $N_{newmivin}$ is the number of method $k \in V_m, k \rightarrow m \in$

$E_m,\ k \to c \in E_{pri} \cup E_{pro} \cup E_{pub},\ m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$

- Number of other methods which invoke the considered method within the same class (N_{mivout}); N_{mivout} is the number of edges from the considered method $m \in V_m \cup V_{mri},\ m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$ to a method $k \in V_m \cup V_{mri},\ k \to c \in E_{pri} \cup E_{pro} \cup E_{pub};\ m \to k \in E_m$
- Number of other new methods which invoke the considered method within the same class ($N_{newmivout}$); $N_{newmivout}$ is the number of method $k \in V_m, m \to k \in E_m,\ k \to c \in E_{pri} \cup E_{pro} \cup E_{pub},\ m \to c \in E_{pri} \cup E_{pro} \cup E_{pub}$
- Number of inherited attributes which are implemented by the considered method in a different way from the overridden method ($N_{diffatt}$); $N_{diffatt}$ is the number of inherited attributes which are accessed by the considered method or other methods called by the considered method to yield the net effect on the inherited attributes not equivalent to those of the inherited attributes of the ancestors made by the overridden method [35].

2. Inheritance: Let m be a method vertex or a referred inherited method vertex and c be a class vertex in a IFG, an AAG, or an MIG. The metrics of method m are described as follows:

   - Signature-operation and Body-operation ($SBOperM$); define/use: Determine whether the vertex tag of the considered method vertex m, $T(m)$, is $\langle D_{si},\ D_{bi}\rangle$, $\langle U_{si},\ D_{bi}\rangle$, or $\langle U_{si},\ U_{bi}\rangle$
   - new/inherited ($M_{nw/ih}$): whether it is newly defined or inherited from a superclass. A method m is newly defined if $T(m) = \langle D_{si},\ D_{bi}\rangle$. Otherwise, the method m is inherited from a superclass, i.e., $T(m) = \langle U_{si},\ D_{bi}\rangle$ or $\langle U_{si},\ U_{bi}\rangle$,

- Number of inherited attribute with signature inheritance define ($\langle D_s, N_b \rangle$) whose values are defined by the considered method (N_{aDsdef}); N_{aDsdef} is the number of inherited attributes $a \in V_{ari}$, $T(a) = \langle D_s, N_b \rangle$, $a \rightarrow m \in E_{da}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$

- Number of inherited attributes with signature inheritance use ($\langle U_s, N_b \rangle$) whose values are defined by the considered method (N_{aUsdef}); N_{aUsdef} is the number of inherited attributes $a \in V_{ari}$, $T(a) = \langle U_s, N_b \rangle$, $a \rightarrow m \in E_{da}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$

- Number of inherited attributes with signature inheritance define ($\langle D_s, N_b \rangle$) whose values are used by the considered method (N_{aDsuse}); N_{aDsuse} is the number of inherited attributes $a \in V_{ari}$, $T(a) = \langle D_s, N_b \rangle$, $a \rightarrow m \in E_{ua}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$

- Number of inherited attributes with signature inheritance use ($\langle U_s, N_b \rangle$) whose values are used by the considered method (N_{aUsuse}); N_{aUsuse} is the number of inherited attributes $a \in V_{ari}$, $T(a) = \langle U_s, N_b \rangle$, $a \rightarrow m \in E_{ua}$ and $\exists\, c \in V_c, a \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$

- Number of other inherited methods with signature and body inheritance define ($\langle D_s, D_b \rangle$) which are invoked by the considered method within the same class ($N_{Dsbmivin}$); $N_{Dsbmivin}$ is the number of methods $k \in V_{mri}$, $T(k) = \langle D_s, D_b \rangle$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$, $k \rightarrow m \in E_m$

- Number of other inherited methods with signature inheritance use and body inheritance define ($\langle U_s, D_b \rangle$) which are invoked by the considered method within the same class ($N_{UsDbmivin}$); $N_{UsDbmivin}$ is the number of methods $k \in V_{mri}$, $T(k) = \langle U_s, D_b \rangle$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in$

$E_{pri} \cup E_{pro} \cup E_{pub},\ k \rightarrow m \in E_m$

- Number of other inherited methods with signature and body inheritance use ($\langle U_s, U_b\rangle$) which are invoked by the considered method within the same class ($N_{Usbmivin}$); $N_{Usbmivin}$ is the number of methods $k \in V_{mri}$, $T(k) = \langle U_s, U_b\rangle$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$, $k \rightarrow m \in E_m$
- Number of other inherited methods with signature and body inheritance use ($\langle U_s, U_b\rangle$) which invoke the considered method within the same class ($N_{Usbmivout}$); $N_{Usbmivout}$ is the number of methods $k \in V_{mri}$, $T(k) = \langle U_s, U_b\rangle$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$, $m \rightarrow k \in E_m$
- Number of other inherited methods with signature and body inheritance define ($\langle D_s, D_b\rangle$) which invoke the considered method within the same class ($N_{Dsbmivout}$); $N_{Dsbmivout}$ is the number of methods $k \in V_{mri}$, $T(k) = \langle D_s, D_b\rangle$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$, $m \rightarrow k \in E_m$
- Number of other inherited methods with signature inheritance use and body inheritance define ($\langle U_s, D_b\rangle$) which invoke the considered method within the same class ($N_{UsDbmivout}$); $N_{UsDbmivout}$ is the number of methods $k \in V_{mri}$, $T(k) = \langle U_s, D_b\rangle$, $k \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$, $m \rightarrow k \in E_m$
- Number of methods with signature and body inheritance define ($\langle D_s, D_b\rangle$) which are members of ancestors and invoked by the considered method ($N_{actDsbmin}$); $N_{actDsbmin}$ is the number of methods $\delta \in V_m \cup V_{mri}$, $T(\delta) = \langle D_s, D_b\rangle$, $\delta \rightarrow m \in E_m$ and $\exists\ \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\ \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$

- Number of methods with signature inheritance use and body inheritance define ($\langle U_s, D_b\rangle$) which are members of ancestors and invoked by the considered method ($N_{actUsDbmin}$); $N_{actUsDbmin}$ is the number of methods $\delta \in V_m \cup V_{mri}$, $T(\delta) = \langle U_s, D_b\rangle$, $\delta \rightarrow m \in E_m$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$
- Number of methods with signature and body inheritance use ($\langle U_s, U_b\rangle$) which are members of ancestors and invoked by the considered method ($N_{actUsbmin}$); $N_{actUsbmin}$ is the number of methods $\delta \in V_m \cup V_{mri}$, $T(\delta) = \langle U_s, U_b\rangle$, $\delta \rightarrow m \in E_m$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$
- Number of attributes with signature inheritance define ($\langle D_s, N_b\rangle$) which are members of ancestors and accessed by the considered method ($N_{actDsatt}$); $N_{actDsatt}$ is the number of attributes $\delta \in V_a \cup V_{ari}$, $T(\delta) = \langle D_s, N_b\rangle$, $\delta \rightarrow m \in E_{da} \cup E_{ua}$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$
- Number of attributes with signature inheritance use ($\langle U_s, N_b\rangle$) which are members of ancestors and accessed by the considered method ($N_{actUsatt}$); $N_{actUsatt}$ is the number of attributes $\delta \in V_a \cup V_{ari}$, $T(\delta) = \langle U_s, N_b\rangle$, $\delta \rightarrow m \in E_{da} \cup E_{ua}$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\exists\, \alpha \xrightarrow[E_{hpub} \cup E_{hpro} \cup E_{hpri}]{} c$
- Number of methods with signature and body inheritance define ($\langle D_s, D_b\rangle$) which are members of parents and invoked by the considered method ($N_{parDsbmin}$); $N_{parDsbmin}$ is the number of methods $\delta \in V_m \cup V_{mri}$, $T(\delta) = \langle D_s, D_b\rangle$, $\delta \rightarrow m \in E_m$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of methods with signature inheritance use and body inheritance define ($\langle U_s, D_b\rangle$) which are members of parents and invoked by the considered method ($N_{parUsDbmin}$); $N_{parUsDbmin}$ is the number of methods $\delta \in$

$V_m \cup V_{mri},\ T(\delta) = \langle U_s, D_b \rangle\ ,\ \delta \rightarrow m \in E_m$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$

- Number of methods with signature and body inheritance use ($\langle U_s, U_b \rangle$) which are members of parents and invoked by the considered method ($N_{parUsbmin}$); $N_{parUsbmin}$ is the number of methods $\delta \in V_m \cup V_{mri},\ T(\delta) = \langle U_s, U_b \rangle\ ,\ \delta \rightarrow m \in E_m$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of attributes with signature inheritance define ($\langle D_s, N_b \rangle$) which are members of parents and accessed by the considered method ($N_{parDsatt}$); $N_{parDsatt}$ is the number of attributes $\delta \in V_a \cup V_{ari}, T(\delta) = \langle D_s, N_b \rangle\ , \delta \rightarrow m \in E_{da} \cup E_{ua}$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Number of attributes with signature inheritance use ($\langle U_s, N_b \rangle$) which are members of parents and accessed by the considered method ($N_{parUsatt}$); $N_{parUsatt}$ is the number of attributes $\delta \in V_a \cup V_{ari}, T(\delta) = \langle U_s, N_b \rangle\ , \delta \rightarrow m \in E_{da} \cup E_{ua}$ and $\exists\, \alpha \in V_c, \delta \rightarrow \alpha \in E_{pub} \cup E_{pro}$ and $\alpha \rightarrow c \in E_{hpub} \cup E_{hpro} \cup E_{hpri}$
- Level of class that originally defines the considered method (L_{odefm}): Let c_1 and c_2 be class vertices in an IFG; $\{c_1,\ c_2\} \subseteq V_c$. Let the considered method m be a member of class c_1, $m \rightarrow c_1 \in E_{pri} \cup E_{pub} \cup E_{pro}$. c_2 is said to be the class that originally defines method m if and only if the following conditions are true:

  (i) $m \rightarrow c_2 \in E_{pub} \cup E_{pro}$ and $T(TV(c_2 \rightarrow m)) = \langle D_{si},\ D_{bi} \rangle$

  (ii) $T(TV(c_1 \rightarrow m)) = \langle U_{si},\ D_{bi} \rangle$ or $\langle U_{si},\ U_{bi} \rangle$

  (iii) $c_2 \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} c_1$ and

  (iv) for all α, $\alpha \in V_c \wedge c_2 \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} \alpha \wedge \alpha \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} c_1$, such that $T(TV(\alpha \rightarrow m)) = \langle U_{si},\ D_{bi} \rangle$ or $\langle U_{si},\ U_{bi} \rangle$

L_{odefm} is the number of classes along the path $c_2 \xrightarrow[E_{hpri} \cup E_{hpro} \cup E_{hpub}]{} c_1$ which includes c_2 but excludes c_1.

3. Association: Let m be a method vertex or a referred inherited method vertex and c be a class vertex in an ASG. The method m is a member of class c; $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$. The metrics of method m are described as follows:

   - Number of invocations via ordinary association (N_{Moriv}): N_{Moriv} is the number of edges from m to k; $m \rightarrow k \in E_m$, $k \in V_m \cup V_{mri}$ and $m \rightarrow c \in E_{pub}$ and $\forall\, \alpha \in V_c$, $k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \xrightarrow[E_{fa}]{} \alpha = \varnothing$ and $c \xrightarrow[E_{fa}]{} k = \varnothing$
   - Number of invocations via friend member function association (N_{Mfmiv}): N_{Mfmiv} is the number of edges from m to k; $m \rightarrow k \in E_m$, $k \in V_m \cup V_{mri}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \alpha \in V_c$, $k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \rightarrow k \in E_{fa}$
   - Number of invocations via friend class association (N_{Mfciv}): N_{Mfciv} is the number of edges from m to k; $m \rightarrow k \in E_m$, $k \in V_m \cup V_{mri}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \alpha \in V_c$, $k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \rightarrow \alpha \in E_{fa}$
   - Number of invocations via friend operation association (N_{Mfoiv}): N_{Mfoiv} is the number of edges from m to k; $m \rightarrow k \in E_m$, $k \in V_{gf}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $c \rightarrow k \in E_{fa}$
   - Number of other methods invoked by the considered method via ordinary association (N_{Masoin}); N_{Masoin} is the number of edges from k to m; $k \rightarrow m \in E_m$, $k \in V_m \cup V_{mri}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\forall \alpha \in V_c$, $k \rightarrow \alpha \in E_{pub}$ and $\alpha \xrightarrow[E_{fa}]{} c = \varnothing$ and $\alpha \xrightarrow[E_{fa}]{} m = \varnothing$
   - Number of other methods invoked by the considered method via friend member function association (N_{Masfin}); N_{Masfin} is the number of edges from k

to m; $k \rightarrow m \in E_m$, $k \in V_m \cup V_{mri}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \alpha \in V_c,\ k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\alpha \rightarrow m \in E_{fa}$

- Number of other methods in other classes invoked by the considered method via friend class association (N_{Mascin}); N_{Mascin} is the number of edges from k to m; $k \rightarrow m \in E_m$, $k \in V_m \cup V_{mri}$ and $m \rightarrow c \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\exists\, \alpha \in V_c,\ k \rightarrow \alpha \in E_{pri} \cup E_{pro} \cup E_{pub}$ and $\alpha \rightarrow c \in E_{fa}$

Fault metrics and associating graphs are depicted in Table 6.9, 6.10, and 6.11.

6.4 Model Deployment

To clarify how to employ the proposed model, an example is presented based on some selected code fragments as demonstrated in Figure 6.17 to Figure 6.23. These code fragments are taken from a simple program which performs as an address book. The process of model deployment are described as follows:

1. **Software modelling: representing software by a set of graphs.** There are four classes from the code example, namely, Contact, CoworkerContact, OtherContact, and Profile as shown in Figure 6.17 to Figure 6.18. The CoworkerContact is a descendant class of Contact and OtherContact of CoworkerContact. The Profile class call method stringCheck() from CoworkerContact. Two inherited methods in class CoworkerContact, namely, setvalue() and stringcheck() are refined as demonstrated in Figure 6.19 to Figure 6.20. The inherited method setvalue() in class OtherContact, invoked by the method add() shown in line 12 of Figure 6.21, can lead to an SVA fault because the method setvalue() has been redefined in the CoworkerContact class. The classes and their members are represented by a series of corresponding graphs, namely, program structure graph (PSG), attribute access

Table 6.9: Object-Oriented metrics and associating graphs

Metric	**Associating graph**				
	PSG	**AAG**	**MIG**	**IFG**	**ASG**
N_{newM}	$\surd$				
N_{newA}	$\surd$				
N_{priA}	$\surd$				
N_{pubA}	$\surd$				
N_{proA}	$\surd$				
N_{priM}	$\surd$				
N_{pubM}	$\surd$				
N_{proM}	$\surd$				
N_{mdiav}	$\surd$				
N_{muiav}	$\surd$				
N_{ihp}				$\surd$	
N_{anc}				$\surd$	
N_{riaa}				$\surd$	
N_{riap}				$\surd$	
N_{ima}				$\surd$	
N_{imp}				$\surd$	
N_{rima}				$\surd$	
N_{rimp}				$\surd$	
N_{DSria}				$\surd$	
N_{DSrim}				$\surd$	
N_{DBrim}				$\surd$	
N_{Urim}				$\surd$	
$N_{iaDsbmin}$				$\surd$	
$N_{iaUsDbmin}$				$\surd$	
$N_{iaUsbmin}$				$\surd$	
$N_{ipDsbmin}$				$\surd$	
$N_{ipUsDbmin}$				$\surd$	
$N_{ipUsbmin}$				$\surd$	
N_{iaDsa}				$\surd$	
N_{iaUsa}				$\surd$	
N_{ipDsa}				$\surd$	
N_{ipUsa}				$\surd$	
N_{par}				$\surd$	
$N_{riaatot}$				$\surd$	
$N_{riaptot}$				$\surd$	
$N_{rimatot}$				$\surd$	
$N_{rimptot}$				$\surd$	
N_{imatot}				$\surd$	
N_{imptot}				$\surd$	
$N_{iaDsbmintot}$				$\surd$	
$N_{iaUsDbmintot}$				$\surd$	
$N_{iaUsbmintot}$				$\surd$	
$N_{ipDsbmintot}$				$\surd$	
$N_{ipUsDbmintot}$				$\surd$	
$N_{ipUsbmintot}$				$\surd$	
$N_{iaDsatot}$				$\surd$	
$N_{iaUsatot}$				$\surd$	
$N_{ipDsatot}$				$\surd$	
$N_{ipUsatot}$				$\surd$	
DIT				$\surd$	

Table 6.10: Object-Oriented metrics and associating graphs (continued)

Metric	**Associating graph**				
	PSG	**AAG**	**MIG**	**IFG**	**ASG**
$N_{ihamintot}$				✓	
$N_{ihpmintot}$				✓	
$N_{ihaatot}$				✓	
$N_{ihpatot}$				✓	
$N_{DSriatot}$				✓	
$N_{DSrimtot}$				✓	
$N_{DBrimtot}$				✓	
$N_{Urimtot}$				✓	
N_{ASOmin}					✓
N_{ASFmin}					✓
N_{ASCmin}					✓
$N_{ASOmout}$					✓
$N_{ASFmout}$					✓
$N_{ASCmout}$					✓
$N_{ASPmout}$					✓
N_{ASOa}					✓
N_{ASFa}					✓
N_{ASCa}					✓
N_{ASPa}					✓
N_{ASFcin}					✓
N_{ASFo}					✓
$N_{ASFcmin}$					✓
N_{ASOcin}					✓
$N_{ASFcout}$					✓
$N_{ASFcmout}$					✓
$N_{ASOcout}$					✓
$AMtype$	✓				
N_{atac}		✓			
$N_{newMdef}$		✓			
$N_{newMuse}$		✓			
$N_{Diffmet}$	✓	✓	✓	✓	✓
$SOperA$				✓	
$A_{nw/ih}$	✓			✓	
$N_{mDsbdef}$		✓		✓	
$N_{mUsDbdef}$		✓		✓	
$N_{mUsbdef}$		✓		✓	
$N_{mDsbuse}$		✓		✓	
$N_{mUsDbuse}$		✓		✓	
$N_{mUsbuse}$		✓		✓	
L_{odefa}				✓	
N_{Aoras}					✓
N_{Afmas}					✓
N_{Afcas}					✓
N_{Afoas}					✓

Table 6.11: Object-Oriented metrics and associating graphs (continued)

Metric	**Associating graph**				
	PSG	**AAG**	**MIG**	**IFG**	**ASG**
$MMtype$	✓				
N_{aacm}		✓			
N_{mivin}			✓		
$N_{newmivin}$			✓		
N_{mivout}			✓		
$N_{newmivout}$			✓		
$N_{diffatt}$	✓	✓	✓	✓	✓
$SBOperM$				✓	
$M_{nw/ih}$	✓			✓	
N_{aDsdef}		✓		✓	
N_{aUsdef}		✓		✓	
N_{aDsuse}		✓		✓	
N_{aUsuse}		✓		✓	
$N_{Dsbmivin}$			✓	✓	
$N_{UsDbmivin}$			✓	✓	
$N_{Usbmivin}$			✓	✓	
$N_{Dsbmivout}$			✓	✓	
$N_{UsDbmivout}$			✓	✓	
$N_{Usbmivout}$			✓	✓	
$N_{actDsbmin}$				✓	
$N_{actUsDbmin}$				✓	
$N_{actUsbmin}$				✓	
$N_{actDsatt}$				✓	
$N_{actUsatt}$				✓	
$N_{parDsbmin}$				✓	
$N_{parUsDbmin}$				✓	
$N_{parUsbmin}$				✓	
$N_{parDsatt}$				✓	
$N_{parUsatt}$				✓	
L_{odefm}				✓	
N_{Moriv}					✓
N_{Mfmiv}					✓
N_{Mfciv}					✓
N_{Mfoiv}					✓
N_{Masoin}					✓
N_{Masfin}					✓
N_{Mascin}					✓

graph (AAG), method invocation graph (MIG), inheritance flow graph (IFG) with inheritance path, association graph (ASG) with association path, as depicted in Figure 6.24 to Figure 6.29. Figure 6.24 shows the structure of three classes, while the detail of attribute access and method invocation within each class is demonstrated in Figure 6.25 to Figure 6.27. The inheritance and association relationships among those classes are represented in Figure 6.28 and Figure 6.29, respectively.

2. **Class selection: selecting classes to consider.** All the classes so implemented will be participating in software fault prediction. The code fragments in Figure 6.22 to Figure 6.23 demonstrate the actual implementation of these classes. Due to class-only category implementation, only its respective inputdata() and checkprofile() functions are represented by control flow graphs in Figure 6.30 to Figure 6.32.

3. **Metric extraction.** Derive metrics of selected classes, attributes, and methods from graphs employing metric definition in Section 6.3.

4. **Fault prediction.** Employ fault predictive models of class with appropriate extracted metrics to predict whether the class is faulty or fault-free. Faulty classes are further explored to locate the cause of faults, faulty methods and attributes of the faulty class are examined using fault predictive models for method and attribute, respectively. Details of the proposed fault prediction algorithm is described in the next section.

6.5 Algorithm

The proposed fault prediction algorithms of this study consist of two phases, namely, fault predictive model construction and fault prediction. In the predictive model con-

```
class    Contact
{
  public:

          char   name[25];
          char   address[50];
          char   telephone[15];
          Category ptype;
          bool   stringCheck(     char  *, int );
          bool   CategoryCheck(      int );
          void   add(   int );
          bool   compare(    char  *);
          void   display();
          void   setvalue(    char  *, int );
 };
```

(a)

```
class CoworkerContact : public Contact
{

public:
          char duty[100];
          void add(int);
          void display();
          void getnote();
          void setvalue(char*,int);
          bool stringCheck(char*,int);
};
```

(b)

Figure 6.17: (a) Code for Contact class. (b) Code for CoworkerContact class.

```
class    OtherContact :       public   CoworkerContact
{


public    :
          void   add(  int );
          void   display();
          void   getnote();

};
```

(a)

```
class    Profile
{
public    :
  char    name[25];
  char    telephone[15];
  char    homeaddress[50];
   char   officeaddress[50];
   void   addprofile();
  bool    checktext(    char  *, int );
};

 bool    Profile::checktext(       char   *tt, int  ft)
 {
  return CoworkerContact::stringCheck(tt,ft);
  }
```

(b)

Figure 6.18: (a) Code for OtherContact class. (b) Code for Profile class.

```
void    Contact::setvalue(      char  * value,   int  n)
{
        int  i;
        if  (n==1)
        {
                for  (i=0;i<25;i++)
                {
                        name[i]= value[i];
                }
        }
        else   if (n == 2)
        {
                for  (i=0;i<50;i++)
                {
                        address[i] = value[i];
                }
        }
        else   if (n==3)
        {
                for  (i=0;i<15;i++)
                {
                        telephone[i]= value[i];
                }
        }

}
```

```
void    CoworkerContact::setvalue(          char  * value,   int  n)
// insert SVA fault
{
        int  i;
        if  (n==3)    // changed from n==1
        {
          for(i=0;i<50;i++)
          {       name[i]=value[i];

          }
        }
        else   if (n == 2)
        {
                for  (i=0;i<50;i++)
                {
                        address[i] = value[i];
                }
        }
        else   if (n==1)
        {
                for  (i=0;i<15;i++)      // changed from n==3
                {
                        telephone[i]= value[i];
                }
        }

}
```

(a) (b)

Figure 6.19: (a) Code for Contact::setvalue(char*,int). (b) Code for CoworkerContact::setvalue(char*,int).

```
bool    Contact::stringCheck(        char  * s,  int  n)
{
        unsigned      int  i;

        if  (n==1)
        {
                if  (strlen(s)==0 || strlen(s) > 25)
                {
                        cout << "\nName cannot be empty or more than 25 characters";
                        return    false  ;
                }
                else
                        return    true  ;
        }

        if  (n==2)
        {
                if  (strlen(s)==0 || strlen(s) > 50)
                {
                        cout << "\nAddress cannot be empty or more than 50 characters";
                        return    false  ;
                }
                else
                        return    true  ;
        }

        if  (n==3)
        {

                for  (i = 0; i < strlen(s); i++)
                {
                        if  (isalpha(s[i]))
                        {
                                cout << "\nAlphabets not allowed";
                                return    false  ;
                        }

                }

                if  (strlen(s)==0 || strlen(s) > 15)
                {
                        cout << "\nTelephone cannot be empty or more than 15 characters\n";
                        return    false  ;
                }
                else
                        return    true  ;
        }

                else
                return    true  ;

}
```

```
bool    CoworkerContact::stringCheck(            char  * s,  int  n)
{
        unsigned      int  i;

        if  (n==1)
        {
                if  (strlen(s)==0 || strlen(s) > 30) // changed from 25
                {
                        cout << "\nName cannot be empty or more than 25 characters";
                        return    false  ;
                }
                else
                        return    true  ;
        }

        if  (n==2)
        {
                if  (strlen(s)==0 || strlen(s) > 50)
                {
                        cout << "\nAddress cannot be empty or more than 50 characters";
                        return    false  ;
                }
                else
                        return    true  ;
        }

        if  (n==3)
        {

                for  (i = 0; i < strlen(s); i++)
                {
                        if  (isalpha(s[i]))
                        {
                                cout << "\nAlphabets not allowed";
                                return    false  ;
                        }

                }

                if  (strlen(s)==0 || strlen(s) > 15)
                {
                        cout << "\nTelephone cannot be empty or more than 15 characters\n";
                        return    false  ;
                }
                else
                        return    true  ;
        }

                else
                return    true  ;

}
```

(a) (b)

Figure 6.20: (a) Code for Contact::stringCheck(char*,int). (b) Code for CoworkerContact::stringCheck(char*,int).

```
void    OtherContact::add(        int  flag)
{       int  g;
        while   (!cin.get()) {};

        if  (flag==1)
        {
                do
                {           cout << "\n Enter name of other person : ";
                            cin.clear();
                            cin.getline(name,25,'\n');
                            strupr(tname);
                            setvalue(name,1);
                } while   (!stringCheck(tname,1));

                do
                {           cout << "\n Enter address of other person : ";
                            cin.getline(address,50,'\n');
                            strupr(address);
                } while  (!stringCheck(address,2));

                do
                {           cout << "\n Enter telephone of other person : ";
                            cin.getline(telephone,15,'\n');
                } while   (!stringCheck(telephone,3));

                do
                {
                            g=3;
                } while   (!CategoryCheck(g));

                getnote();
        }
        else
        {
                do
                {           cout << "\n Enter new address of other person : ";
                            cin.getline(address,50,'\n');
                            strupr(address);
                } while  (!stringCheck(address,2));

                do
                {           cout << "\n Enter new telephone of other person : ";
                            cin.getline(telephone,15,'\n');
                } while   (!stringCheck(telephone,3));

                getnote();
        }
}
```

Figure 6.21: Code for OtherContact::add()

```
void    inputData(     char   ch)
{
        char   continueAdding='Y';


        if  (ch=='1')
        {
                  ..........

        }
        else   if (ch == '2')
 {
         CoworkerContact cperson;
         ofstream dataIn("addressBookc.dat",ios::ate);                    //create binary file

         do
         {
                  system("CLS");
                  cout << "New coworker person entry";
                  cout << "\n-----------------------\n";
                  cperson.add(1);

                  dataIn.write((      char  *) (&cperson),      sizeof    (cperson));

                  cout << "\nDo you want to add another coworker person? [Y/N] :";
                  cin >>continueAdding;
                  continueAdding = toupper(continueAdding);

         } while   (continueAdding!='N');

         dataIn.close();       //close file stream
        }
        else   if (ch == '3')
 {
         OtherContact operson;
         ofstream dataIn("addressBooko.dat",ios::ate);                    //create binary file

         do
         {
                  system("CLS");
                  cout << "New other person entry";
                  cout << "\n-----------------------\n";
                  operson.add(1);

                  dataIn.write((      char  *) (&operson),      sizeof    (operson));

                  cout << "\nDo you want to add another other person? [Y/N] :";
                  cin >>continueAdding;
                  continueAdding = toupper(continueAdding);

         } while   (continueAdding!='N');

         dataIn.close();       //close file stream
        }
}
```

Figure 6.22: Code fragment for InputData() function

```
void searchData(char ch)
{
        char string[45];
        int found = 0;

        system("CLS");
        cout << "\n\nSearch contact person ";
        cout << "\n--------------------------";

        cout << "\n Enter name :";

        while (!cin.get())
        {  .....      }
        cin.getline(string,45);
        strupr(string);

        ..................

        Checkprofile(string);

        ..................


 }


void CheckProfile{char* persondat}
{
        bool chkresult1;
        bool chkresult2;
        bool chkresult3;

        ..................

        Profile oprofile;

        ..................

        chkresult1 = oprofile.checktext(persondat,1);
        chkresult2 = oprofile.checktext(persondat,2);
        chkresult3 = oprofile.checktext(persondat,3);

        ..................

}
```

Figure 6.23: Code fragment for SearchData() function

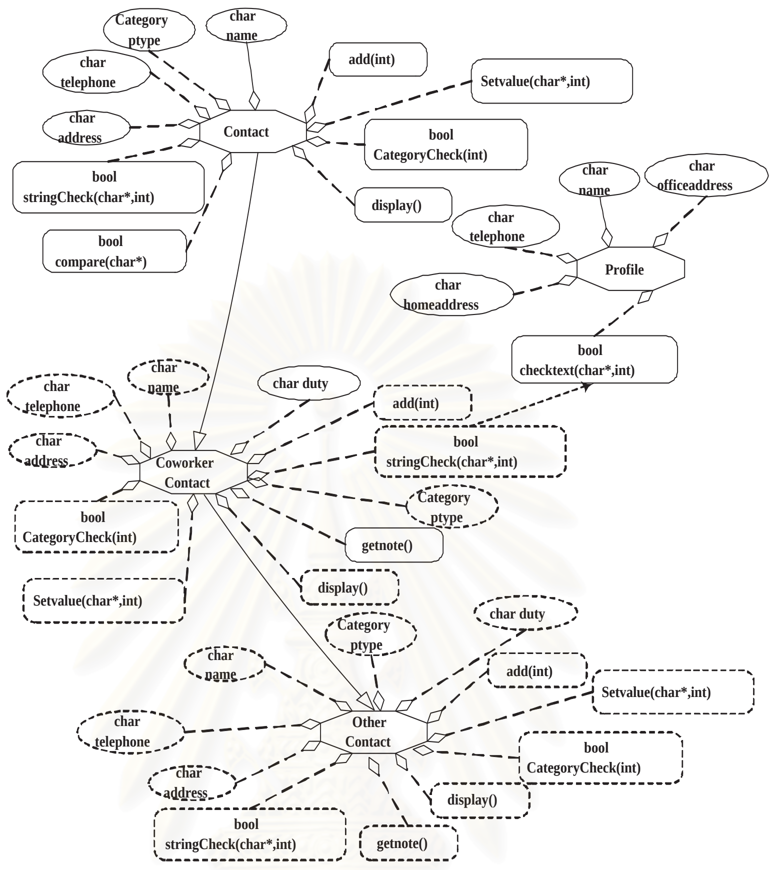


Figure 6.24: Program structure graph (PSG)

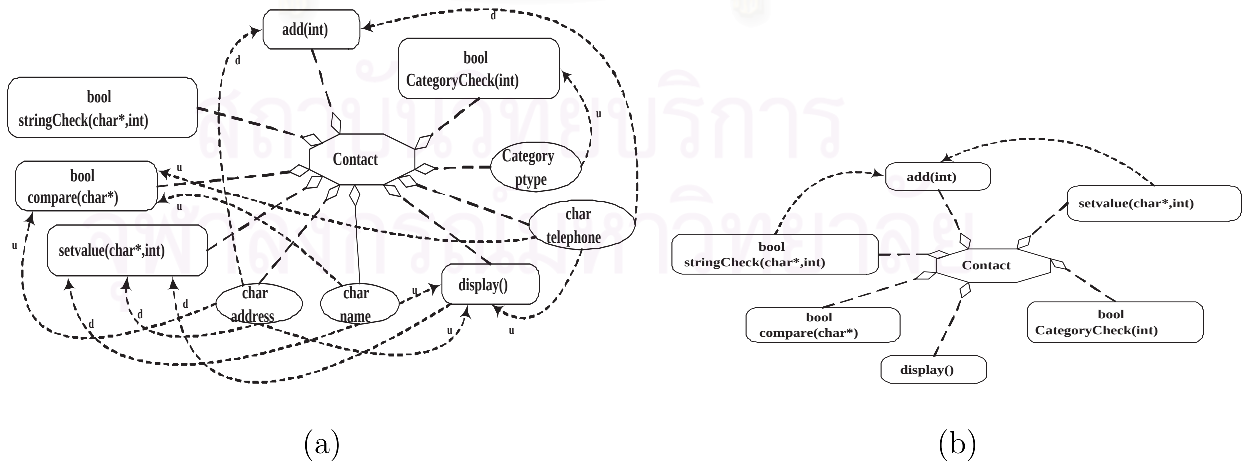


Figure 6.25: (a) Attribute access graph (AAG) for Contact class. (b) Method invocation graph (MIG) for Contact class.

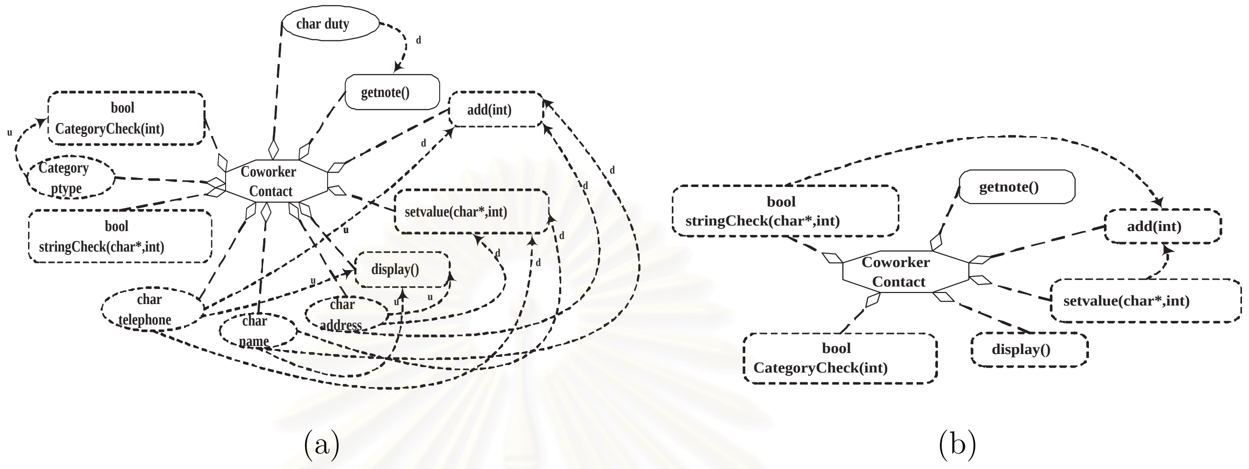


Figure 6.26: (a) Attribute access graph (AAG) for CoworkerContact class. (b) Method invocation graph (MIG) for CoworkerContact class.

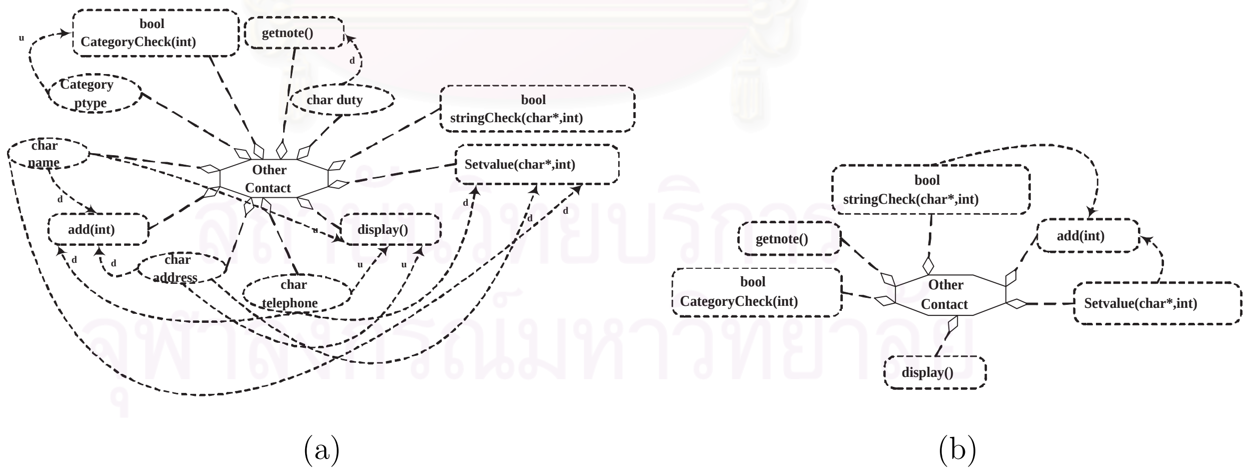


Figure 6.27: (a) Attribute access graph (AAG) for OtherContact class. (b) Method invocation graph (MIG) for OtherContact class.

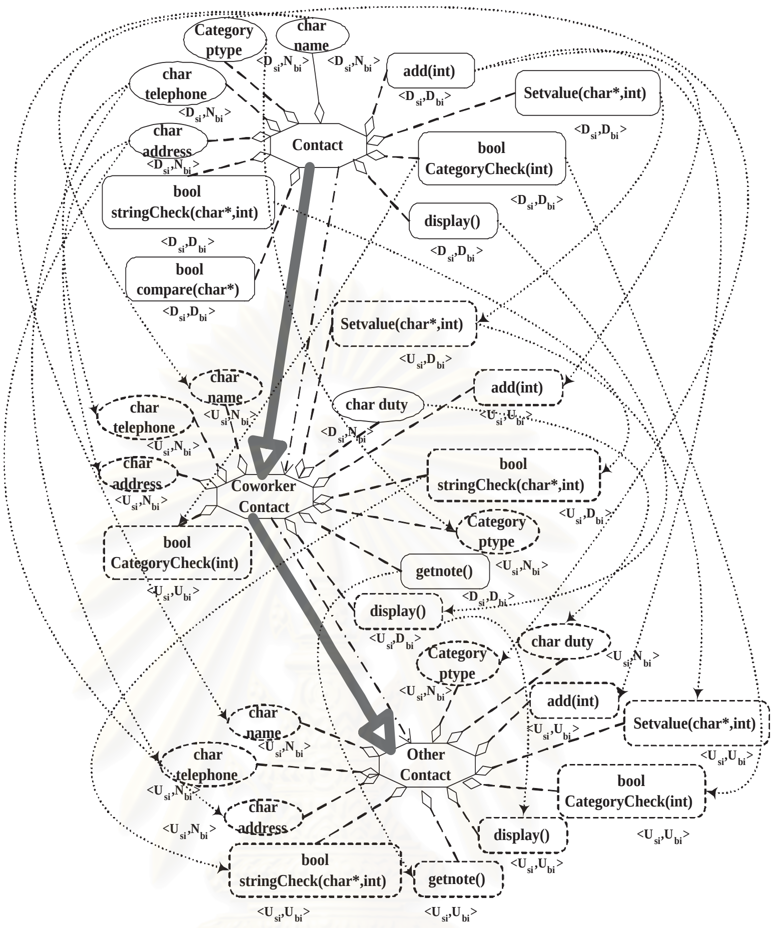


Figure 6.28: Inheritance flow graph (IFG) with inheritance path

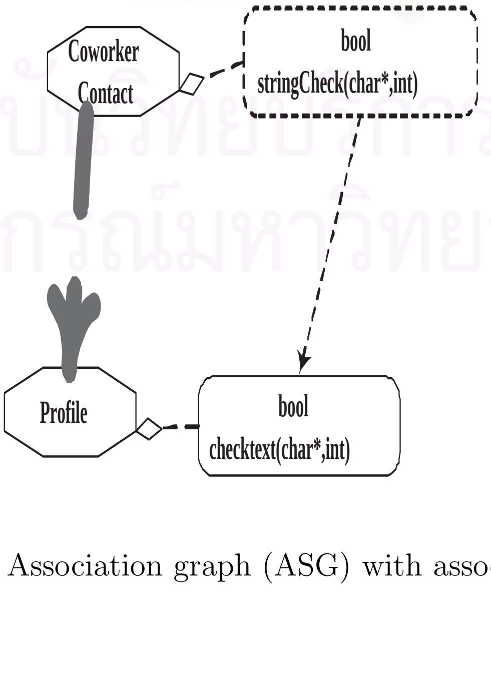
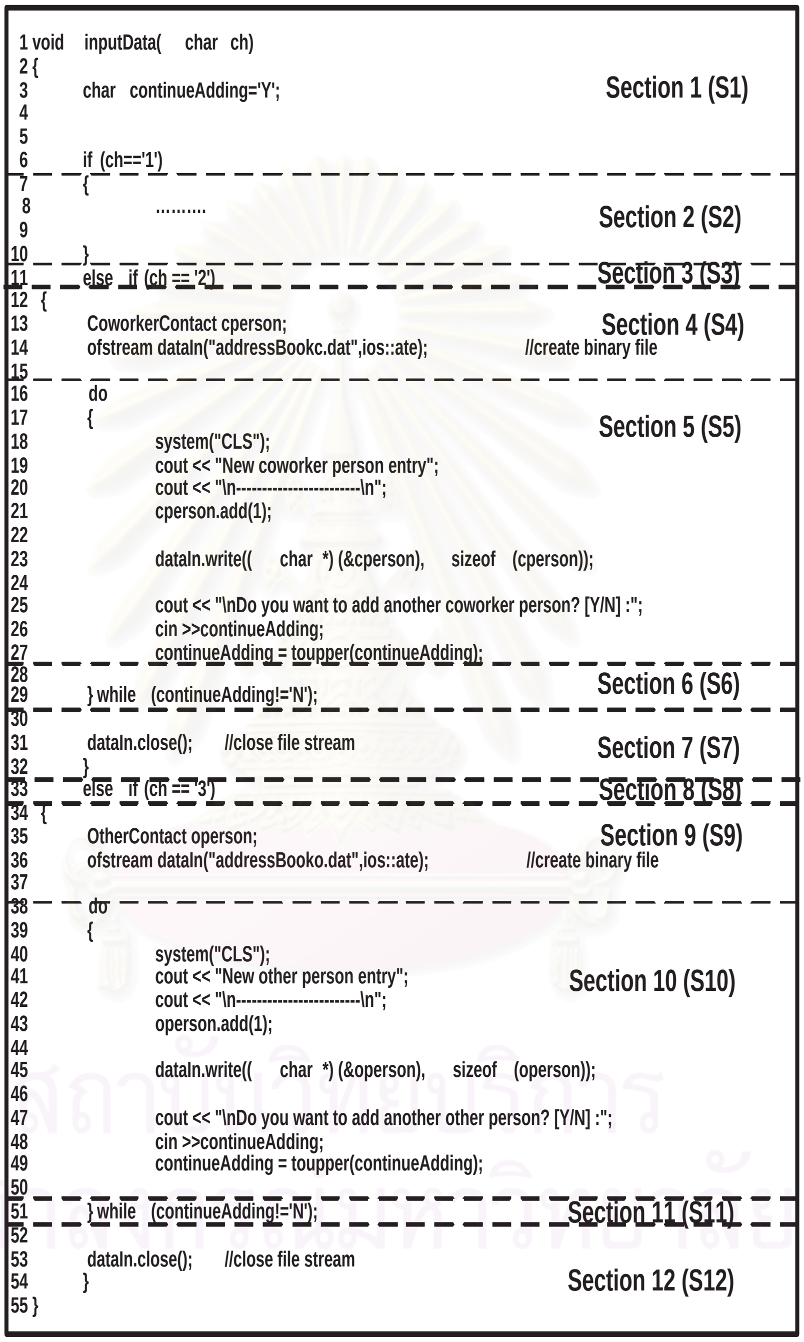


Figure 6.29: Association graph (ASG) with association path



```
void    inputData(    char   ch)
{
        char   continueAdding='Y';


        if  (ch=='1')
        {
                   ..........

        }
        else   if  (ch == '2')
 {
         CoworkerContact cperson;
         ofstream dataIn("addressBookc.dat",ios::ate);                //create binary file

         do
         {
                   system("CLS");
                   cout << "New coworker person entry";
                   cout << "\n-----------------------\n";
                   cperson.add(1);

                   dataIn.write((     char  *) (&cperson),     sizeof    (cperson));

                   cout << "\nDo you want to add another coworker person? [Y/N] :";
                   cin >>continueAdding;
                   continueAdding = toupper(continueAdding);

         } while   (continueAdding!='N');

         dataIn.close();      //close file stream
        }
        else   if  (ch == '3')
 {
         OtherContact operson;
         ofstream dataIn("addressBooko.dat",ios::ate);                //create binary file

         do
         {
                   system("CLS");
                   cout << "New other person entry";
                   cout << "\n-----------------------\n";
                   operson.add(1);

                   dataIn.write((     char  *) (&operson),     sizeof    (operson));

                   cout << "\nDo you want to add another other person? [Y/N] :";
                   cin >>continueAdding;
                   continueAdding = toupper(continueAdding);

         } while   (continueAdding!='N');

         dataIn.close();      //close file stream
        }
}
```

Figure 6.30: Code fragment for InputData() function with divided sections

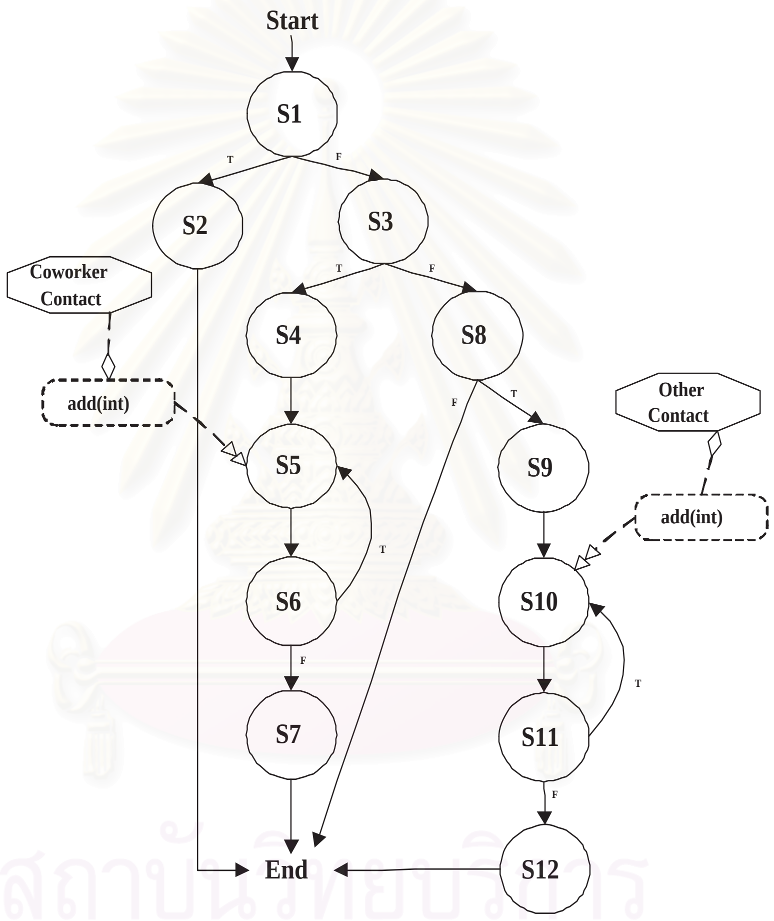
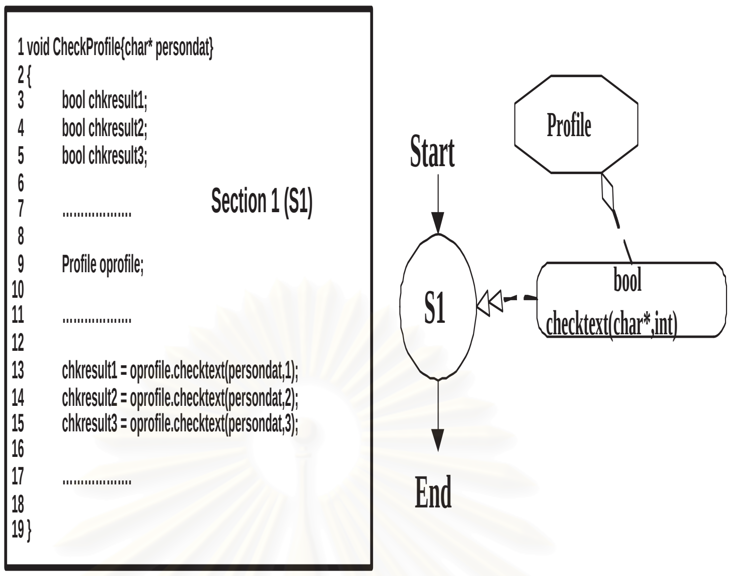


Figure 6.31: Control flow graph (CFG) for InputData function



```
void CheckProfile{char* persondat}
{
        bool chkresult1;
        bool chkresult2;
        bool chkresult3;

        ....................

        Profile oprofile;

        ....................

        chkresult1 = oprofile.checktext(persondat,1);
        chkresult2 = oprofile.checktext(persondat,2);
        chkresult3 = oprofile.checktext(persondat,3);

        ....................

}
```

Figure 6.32: Control flow graph (CFG) for CheckProfile function

struction, neural network techniques are employed to construct three models which are used for fault prediction in class, method, and attribute, respectively. Then the predictive models are applied to classes, methods, and attributes to detect faultiness in the fault prediction phase. Fault prediction results are analyzed to find the cause and location of faults.

6.5.1 Fault predictive model construction

The predictive models were constructed using multilayer perceptron (MLP) with backpropagation learning algorithm [50]. Three MLP models were constructed, namely, class model, method model, and attribute model, to represent faults in class, method, and attribute, respectively. The objective of separating the model representation is to categorically determine if class, method, and attribute are faulty or fault-free within each model. The structure of each predictive model encompasses input, hidden, and output layers, where by measurement metrics of class, method, and attribute in Section 6.3 are

Table 6.12: Selected metrics employed as input of predictive models

No.	Class's fault predictive model from [4, 5]	Class's fault predictive model	Method's fault predictive model	Attribute's fault predictive model
1	$CountDeclInstanceVariableProtected$	N_{pubA}	N_{mivin}	$SOperA$
2	$CountDeclMethodProtected$	N_{proA}	N_{mivout}	$A_{nw/ih}$
3	ECE	N_{priM}	$M_{nw/ih}$	N_{Aoras}
4	ECR	N_{proM}	N_{Moriv}	N_{Afmas}
5	$ImRef$	N_{ASOmin}	N_{Mfmiv}	N_{Afcas}
6	$DiffDeff$	N_{ASFmin}	N_{Mfciv}	N_{Afoas}
7	$DiffOvrrl$	N_{ASCmin}	N_{Mfoiv}	$N_{newMdef}$
8	$DepIV$	$N_{ASOmout}$	N_{Masoin}	$N_{newMuse}$
9	$RIpriV$	$N_{ASFmout}$	N_{Masfin}	$N_{mDsbdef}$
10	NOD	$N_{ASCmout}$	N_{Mascin}	$N_{mUsDbdef}$
11	NOC	$N_{ASPmout}$	$N_{newmivin}$	$N_{mDsbuse}$
12		N_{ASOa}	$N_{newmivout}$	$N_{mUsDbuse}$
13		N_{ASFa}	$N_{diffatt}$	$N_{mUsbuse}$
14		N_{ASCa}	N_{aDsdef}	$N_{Diffmet}$
15		N_{ASPa}	N_{aUsdef}	
16		N_{ASFcin}	N_{aDsuse}	
17		N_{ASFo}	N_{aUsuse}	
18		$N_{ASFcmin}$	$N_{Dsbmivin}$	
19		N_{ASOcin}	$N_{UsDbmivin}$	
20		$N_{ASFcout}$	$N_{Usbmivin}$	
21		$N_{ASFcmout}$	$N_{Usbmivout}$	
22		$N_{ASOcout}$	$N_{Dsbmivout}$	
23		$N_{DSriatot}$	$N_{UsDbmivout}$	
24		$N_{DSrimtot}$	$N_{actDsbmin}$	
25		N_{mdiav}	$N_{actUsDbmin}$	
26		N_{muaiv}	$N_{actUsbmin}$	
27		$N_{iaUsDbmintot}$	$N_{actDsatt}$	
28		$N_{ipUsDbmintot}$	$N_{actUsatt}$	
29		$N_{ipUsbmintot}$	$N_{parDsbmin}$	
30		$N_{iaDsatot}$	$N_{parUsDbmin}$	
31		$N_{iaUsatot}$	$N_{parUsbmin}$	
32		$N_{ipDsatot}$	$N_{parDsatt}$	
33		$N_{ipUsatot}$	$N_{parUsatt}$	

selected as input of the models using the metric selection algorithm [4]. The expected output value computed from the output node of each model would be zero for the fault-free class (method or attribute) and one for the faulty class (method or attribute). Three selected metric sets for three fault predictive models are listed in Table 6.12.

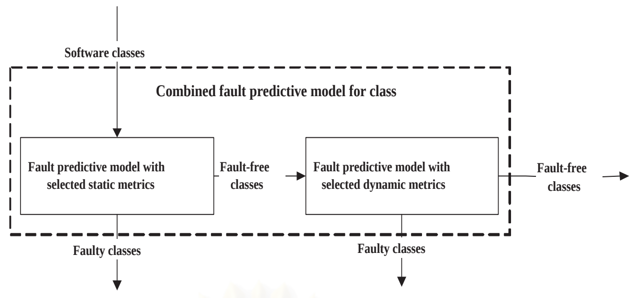

Software classes
Combined fault predictive model for class
Fault predictive model with selected static metrics
Fault-free classes
Fault predictive model with selected dynamic metrics
Fault-free classes
Faulty classes
Faulty classes

Figure 6.33: Diagram of fault predictive model for class.

The configuration of fault predictive model for class is made up of 33 input nodes in input layer, 15 hidden nodes in hidden layer, and 1 output node in output layer. This is an extension of the static predictive model employed in [4, 5] having 11 input nodes in input layer, 5 hidden nodes in hidden layer, and 1 output node in output layer. Fault prediction was performed in two stages, namely, static and dynamic fault predictions. The input is a collection of software classes are first fed into the static fault predictive model. This model classifies the input software classes into faulty and fault-free classes. Then the fault-free classes are further passed onto the dynamic fault predictive model. In so doing, the misclassified faulty classes which are predicted as fault-free classes by the static fault predictive model are procedurally detected. The process of class fault predictive model is shown in Figure 6.33.

Similarly, the fault predictive model for method and attribute consists of 33 input nodes in input layer, 15 hidden nodes in hidden layer, and 1 output node in output layer, 14 input nodes in input layer, 5 hidden nodes in hidden layer, and 1 output node in output layer, respectively. The model construction was carried out in the same manner as that of class.

6.5.2 Fault prediction

A comprehensive two-stage fault predictive algorithm is proposed in this study to pinpoint the cause and location of faults in the participating classes, methods, and attributes. The following procedures elaborate the operation of the algorithm.

1. Select classes, methods, and attributes to participate in a two-stage process, namely, coarse-grained and fine-grained stage.

   (1) Coarse-grained stage: A control flow graph (CFG), attribute access graph (AAG), method invocation graph (MIG), and inheritance flow graph (IFG) of the investigated system are employed. All classes, methods, and attributes which appear in the CFG are chosen. The methods called by the considered methods and the methods calling the considered methods in MIG are also included, along with the attributes accessed by the considered methods that meet the definition given below.

   Definition 6.19: Let m and a be a method and an attribute in an MIG and an AAG. Let V_{select} be a set of selected classes, methods, and attributes. The method m and attribute a will be considered if the following conditions are true:

   i. $\exists\, \delta \in (V_m \cup V_{mri} \cup V_a \cup V_{ari}) \cap V_{select},\ m \rightarrow \delta \in E_m,\ \delta \rightarrow m \in E_m,$ $a \rightarrow m \in E_{da} \cup E_{ua}$

   In the IFG, superclasses or ancestors, methods, and attributes which are inherited by the considered classes, methods, and attributes from the CFG, AAG, and MIG are defined as follows.

Definition 6.20: Let c, m, and a be a class, a method or referred inherited method, and an attribute or referred inherited attribute in an IFG, respectively. The method m and the attribute a are members of class c. Let V_{select} be a set of selected classes, methods, and attributes. The class c, method m, and attribute a will be considered if the following conditions are true:

i. $\exists\, \alpha \in V_c \cap V_{select},\ c \xrightarrow[IHF]{} \alpha$ and

ii. $\exists\, \delta \in (V_m \cup V_{mri} \cup V_a \cup V_{ari}) \cap V_{select},\ m \rightarrow \delta \in E_{mh},\ a \rightarrow \delta \in E_{mh}$

(2) Fine-grained stage: An association graph (ASG) is employed to further investigate the classes, methods, and attributes which relate in some forms of association to the selected classes, methods, and attributes from the coarse-grained stage.

Definition 6.21: Let c, m, and a be a class, a method or referred inherited method, and an attribute or referred inherited attribute in an association graph, respectively. The method m and the attribute a are members of class c. The class c, method m, and attribute a will be considered if the following conditions are true:

i. $\exists\, \alpha \in V_c \cap V_{select},\ c \xrightarrow[ASF]{} \alpha$ and

ii. $\exists\, \delta \in (V_m \cup V_{mri} \cup V_a \cup V_{ari}) \cap V_{select},\ m \rightarrow \delta \in E_m,\ \delta \rightarrow m \in E_m,\ a \rightarrow \delta \in E_{da} \cup E_{ua}$

The selected classes, methods, and attributes from example graphs from Figure 6.4 through 6.16 are depicted in Table 6.13.

2. Apply the selected classes, methods, and attributes to the predictive models.

3. Find the cause and location of fault using sensitivity analysis and graphs.

Table 6.13: Selected classes from coarse-grained and fine-grained stages

Coarse-grained stage		**Fine-grained stage**
CFG	**IFG**	**ASG**
C	A	
D	A,B	A,C
E		A,B
F	A,B,D	

Table 6.14: Selected methods from coarse-grained and fine-grained stages

Coarse-grained stage			**Fine-grained stage**
CFG	**MIG**	**IFG**	**ASG**
C:inival()			
D:setd1()			
D:display()	D:displaychar(char),D:displayint(int)		
E:plusval()			A:plus(int),B:display(int)
F:setb2(float)			
F:setd1(float)			
F:compute()	F:set(int)		
F:write()		A:write()	

Table 6.15: Selected attributes from coarse-grained and fine-grained stages

Coarse-grained stage			**Fine-grained stage**
CFG	**AAG**	**IFG**	**ASG**
D:d2,D:b2	D:b2,D:a1,D:d1,D:d2	A:a1	
E:e3,E:e2	E:e1,E:e2,E:e3		
F:d1,F:b2	F:a1,F:b2,F:f1,F:d1	A:a1,D:d1,B:b2	

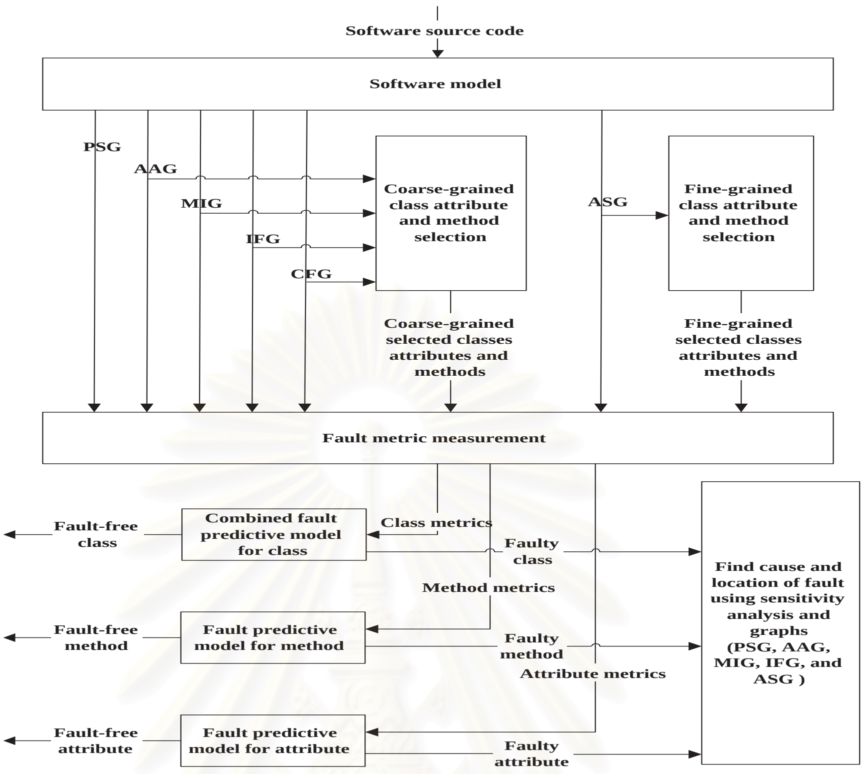

Software source code
Software model
PSG
AAG
MIG
IFG
CFG
Coarse-grained class attribute and method selection
ASG
Fine-grained class attribute and method selection
Coarse-grained selected classes attributes and methods
Fine-grained selected classes attributes and methods
Fault metric measurement
Fault-free class
Combined fault predictive model for class
Class metrics
Faulty class
Method metrics
Fault-free method
Fault predictive model for method
Faulty method
Attribute metrics
Find cause and location of fault using sensitivity analysis and graphs (PSG, AAG, MIG, IFG, and ASG)
Fault-free attribute
Fault predictive model for attribute
Faulty attribute

Figure 6.34: The two-stage fault predictive algorithm process

Figure 6.34 summarizes the process of the proposed algorithm.

6.6 Sensitivity Analysis

As parametric modeling is being applied to solve real world problems, the inherent effects of parameter value change are crucial for the execution of the model. Unfortunately, it is difficult to identify which input parameters actually have influence on the output of model. Sensitivity Analysis (SA) is an approach designed to serve the purpose. The technique estimates the rate of change in the output of a model caused by the changes of the model inputs [51]. If a tiny change of an input leads to great changes in the output, the model is highly sensitive to that input [52]. Thus, the aims of sensitivity analysis are [53]:

- Find the parameters which have great effect on the outputs of a model;
- Study the possible changes of those parameters;
- Determine how those parameters affect the final decision-making; and
- Identify what activities will lighten the effects.

There are many sensitivity analysis methods which can be divided into three categories [54].

1) Mathematical methods

   Mathematical methods assess sensitivity of a model output to the range of variation of an input. They can assess the impact of range of variation in the input variables on the output, so they can be helpful in finding the most important inputs. Mathematical methods include nominal range sensitivity analysis, break-even analysis, difference-in-log-odds ratio, and automatic differentiation.

2) Statistical methods

   Statistical methods involve conducting simulations in which inputs are assigned probability distributions and assessing the impact of variation in inputs on the output distribution. Depending on the method, one or more inputs are varied at a time. Statistical methods allow identifying the effect of interactions among multiple inputs. Statistical methods include regression analysis, analysis of variance, response surface methods, fourier amplitude sensitivity test, and mutual information index.

3) Graphical methods

   Graphical methods give representation of sensitivity in the form of graphs or charts. Generally, graphical methods are used to give visual indication of how an output is affected by variation in inputs. Graphical methods can be quite helpful before further analysis of a model or to describe complex dependencies between inputs and outputs. Graphical methods can also be used to complement the results of mathematical and statistical methods.

In this study, a mathematical sensitivity analysis technique from the mathematical category was employed. The technique was developed by Zurada, Malinowski and Cloete [55] based on neural network model. They first defined a metric for the individual input-output sensitivities, and then showed how this metric could be used to determine the sensitivities of each output over the entire training set. This technique is described as below.

6.6.1 Input-output sensitivities

Only a three layer network is adequate to explain the way to find input-output sensitivities, although the extension of sensitivity analysis to neural networks with more layers is straightforward.

Consider a three layer feedforward neural network, where $\vec{z} = (z_1, \ldots, z_i, \ldots, z_I)$, $\vec{y} = (y_1, \ldots, y_j, \ldots, y_J)$ and $\vec{o} = (o_1, \ldots, o_k, \ldots, o_K)$, denote the input, hidden, and output layers, respectively. Define a training pair p as the tuple $p = (\vec{z}^{(p)}, \vec{t}^{(p)})$, where $\vec{t} = (t_1, \ldots, t_k, \ldots, t_K)$ denotes as the target vector. Given any training pair p, define the sensitivity S_{ki}^p of a trained output o_k with respect to an input z_i as

$$S_{ki}^p = \frac{\partial o_k}{\partial z_i} = \acute{o}_k \sum_{j=1}^{J} w_{kj} \frac{\partial y_j}{\partial z_i} = \acute{o}_k \sum_{j=1}^{J} w_{kj} \acute{y}_j v_{ji} \tag{6.1}$$

where y_j denotes the output of the j-th hidden neuron of the hidden layer $\vec{y}$, $\acute{o}_k$ is the value of the derivative of the output layer activation function

$$o_k = f(\sum_{j=1}^{J} w_{kj} y_j) \tag{6.2}$$

and $\acute{y}_j$ is the value of the derivative of the hidden layer activation function

$$y_j = f(\sum_{i=1}^{I} v_{ji} z_i) \tag{6.3}$$

where w_{kj} denotes the weight value between hidden neuron y_j and output o_k, and v_{ji} is the weight value between input z_i and hidden neuron y_j.

Equation (6.1) only defines the sensitivity of one output o_k with respect to one input z_i for pattern p. For training pattern p, define the pattern sensitivity matrix $S^{(p)}$ which consists of entries S^p_{ki}, as

$$S^{(p)} = \acute{O} W \acute{Y} V \tag{6.4}$$

where $W(K \times J)$ and $V(J \times I)$ are respectively the output and hidden layer weight matrices, and $\acute{O}(K \times K)$ and $\acute{Y}(J \times J)$ are defined as

$$\acute{O} \approx diag(\acute{o}_1, \ldots, \acute{o}_K) \tag{6.5}$$

$$\acute{Y} \approx diag(\acute{y}_1, \ldots, \acute{y}_J) \tag{6.6}$$

6.6.2 Sensitivity measures over entire training set

Equation (6.5) defines the sensitivity matrix for a specific training pattern p. However, each training pair p produces a different sensitivity matrix $S^{(p)}$. In order to apply sensitivity analysis, the sensitivity matrix $S^{(p)}$ must be evaluated over the entire training set. Consequently, three different metrics over the entire training set are defined as follows [55]:

- The mean square average sensitivity matrix S_{avg}

$$S_{ki,avg} \approx \sqrt{\frac{\sum_{p=1}^{P}\left[S_{ki}^{(p)}\right]^2}{P}} \tag{6.7}$$

- The absolute value average sensitivity matrix S_{abs}

$$S_{ki,abs} \approx \frac{\sum_{p=1}^{P}\left|S_{ki}^{(p)}\right|}{P} \tag{6.8}$$

- The maximum sensitivity matrix S_{max}

$$S_{ki,max} \approx max_{p=1,\ldots,P}\left\{S_{ki}^{(p)}\right\} \tag{6.9}$$

Any one of the sensitivity measure matrices above is sufficient to assess the relative significance of each input to each output. In addition, inputs and outputs are needed to be scaled to the same range in order to allow accurate comparison among inputs. If the original inputs and outputs are not scaled to the same range before training, the scaling equation (6.10) has to be applied.

$$S_{ki,avg} = S_{ki,avg}\frac{\left(max_{p=1,\ldots,P}\left\{z_i^{(p)}\right\} - min_{p=1,\ldots,P}\left\{z_i^{(p)}\right\}\right)}{\left(max_{p=1,\ldots,P}\left\{o_k^{(p)}\right\} - min_{p=1,\ldots,P}\left\{o_k^{(p)}\right\}\right)} \tag{6.10}$$

Sensitivity analysis has been applied in various fields, such as marketing. SA is employed to discover important variables that influence sales performance of color television

(CTV) sets in Singapore market, civil engineering [56]. The significance of independent factors for concreting productivity are explored using SA [57], Software Engineering; SA is applied to determine which component affects the reliability of the system most [58]. Software measures are ranked to identify the best software reliability indicators and changes in the system behavior due to changes in system parameters or variables are computed using SA [59, 60].

6.7 Experiment

The experiment was carried out using 560 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 combinations of experimental data provide a good mixture necessary for obtaining general predictive models.

For the purpose of this study, faults have been inserted to 320 classes according to syntactic patterns in [3] while the remaining 240 classes are assumed fault-free. All classes were measured by 55 metrics described in Section 6.3 and 11 metrics/parameters from [4]. There were 790 methods and 760 attributes collected for this experiment. All methods and attributes were measured by 40 metrics and 19 metrics from Section 6.3 and Section 6.3, respectively.

The values of class metrics, method metrics, and attribute metrics are normalized to 0 and 1, and randomly grouped into training set and test set for each predictive models in Section 6.5.1. The training sets were used to trained the predictive models and the test sets were applied to test the models. The predictive models were evaluated by means of some measurement criteria [7] as follows:

- **Type I error (T1):** This error occurs when a faulty class is classified as fault-free.

Table 6.16: Results from fault predictive models

Criterion	**Fault predictive model**		
	Class	**Method**	**Attribute**
Correctness percentage	94.64%	92.40%	91.45%
Type I error (T1)	3.57%	4.43%	5.26%
Type II error (T2)	1.78%	3.16%	3.29%
Quality achieved (C)	94.20%	90.00%	84.61%
Inspection (I)	59.82%	43.67%	33.55%
Waste Inspection (WI)	2.98%	7.35%	9.80%

- **Type II error (T2):** This error occurs when a fault-free class is classified as faulty.
- **Quality achieved (C):** If all faulty classes are properly classified, defects will be removed by extra verification.
- **Inspection (I):** Inspection measures the overall verification cost by considering the percentage of classes that should be verified.
- **Waste Inspection (WI):** Waste inspection is the percentage of classes that do not contain faults but are verified because they have been classified incorrectly.

Table 6.16 shows the result from model testing and evaluation.

By applying the algorithm of fault prediction in Section 6.5.2, a set of selected 40 classes are considered whether they are faulty or fault-free. Thirty-eight classes or 95.0% of the set are correctly predicted by the predictive model. Two fault-free classes are predicted as faulty classes while the rest 20 classes and 18 classes are correctly predicted as faulty and fault-free classes, respectively. Fifty-one methods and 63 attributes involved those selected classes are determined to find which ones are faulty by applying the respective method's predictive model and attribute's predictive model. Five faulty methods are incorrectly predicted as fault-free with 90.19% of accuracy. The attribute's predictive model can predict the faultiness of those selected attributes with 90.48% of accuracy, where 2 faulty attributes and 4 fault-free attributes are incorrectly predicted.

To find the significance of each input parameter of the fault predictive models, partial-derivative sensitivity measure as mentioned in Section 6.6 is applied. For this study, the absolute average sensitivity is computed and demonstrated by graphs in Figure 6.35 to Figure 6.38. Consider the class predictive models, the 9th metric ($RIpriV$) in Figure 6.35 and the 28th metric ($N_{ipUsDbmintot}$) in Figure 6.36 have the highest sensitivity value, which means that the number of methods with signature inheritance use and body inheritance define which are members of parents and invoked by the considered class. The outcomes also indicate the number of refining methods in the ancestor classes that inherited to the descendant class [5] has highest effect on the faultiness of class. Figure 6.37 shows that the 13th metric ($N_{diffatt}$) has the most effect on faultiness of method. By the same token, faultiness of attribute is highly dependent on the third metric (N_{Aoras}) which yields the highest sensitivity value as depicted in Figure 6.38.

When a faulty class is identified, the location of fault is determined by tracing the faulty method (or attribute) which can be reached from the considered faulty class through a sequence of edges from the graphs (PSG, AAG, MIG, IFG, and ASG). Then the cause of fault is determined based on the methods or attributes of that class and all the associated metrics. The order of computation starts from the highest sensitivity metric yield to the lowest one. A guide line for locating fault is established as the followings:

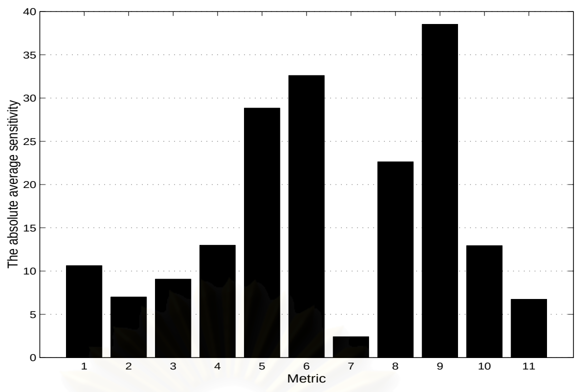


Figure 6.35: The absolute average sensitivity of input for class's predictive model constructed based on metrics from [4, 5]

- Pass a class to the class fault predictive model
- Apply methods and attributes of the faulty class to the method and attribute predictive models, respectively
- The faulty methods and attributes are the fault location of the faulty class
- Employ sensitivity analysis to fault predictive models for class, method, and attribute.
- The metric (input of the fault predictive model) with highest sensitivity value has the most effect on fault

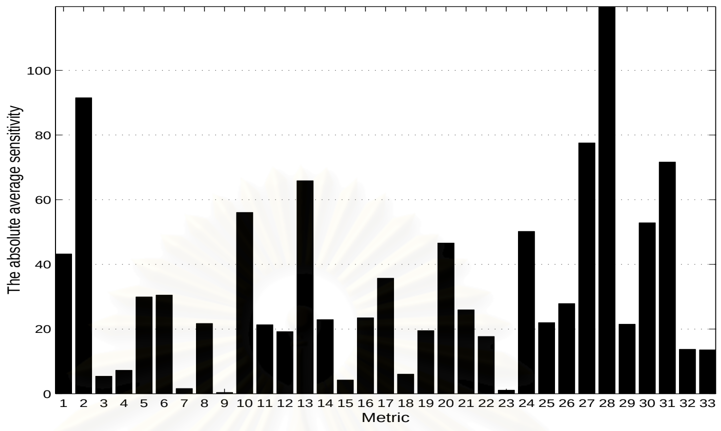


Figure 6.36: The absolute average sensitivity of input for class's predictive model constructed based on the proposed selected class metrics

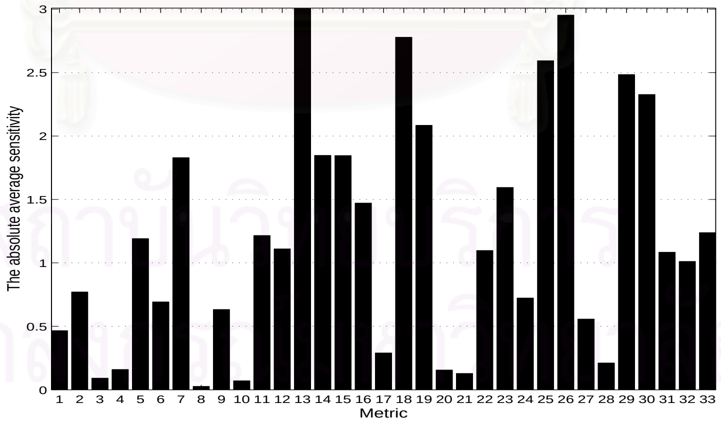


Figure 6.37: The absolute average sensitivity of input for method's predictive model constructed based on the proposed selected method metrics

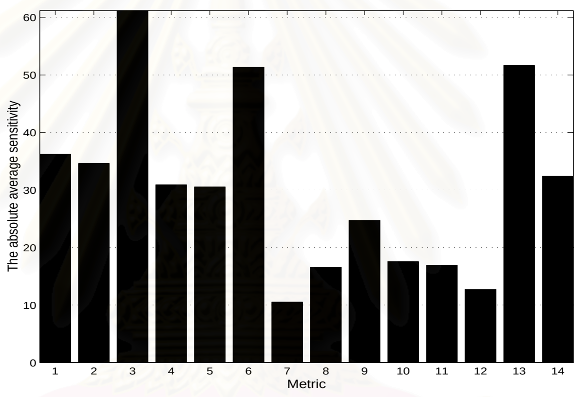


Figure 6.38: The absolute average sensitivity of input for attribute's predictive model constructed based on the proposed selected attribute metrics

CHAPTER VII

CONCLUSION

7.1 Discussion

The proposed software fault prediction approach can predict fault-proneness in both procedural and object-oriented software using faultiness predictive models. The procedural software model yields 83% accuracy while over 90% accuracy is obtained by the object-oriented software model. The reason behind such high yielding results is due to the application of a methodical two-stage process that selects proper metrics to predict and identify fault type.

However, there are faulty software classes being predicted as fault-free because they do not incorporate compile-time or static faults, but possess various recalcitrant dynamic faults. Thus, a set of analytical graphs were employed to accommodate appropriate metrics for dynamic fault prediction, namely, program structure graph (PSG), inheritance flow graph (IFG), attribute access graph (AAG), method invocation graph (MIG), control flow graph (CFG), and association graph (ASG). A dynamic class faultiness predictive model is introduced to assess such faults. Moreover, the locations of faults are also investigated by finding faulty methods and attributes with 92.40% and 91.45% accuracy, respectively.

Prediction results were further analyzed by sensitivity analysis technique to determine the impact of various fault type parameters. Class fault type was fairly accurate in all aspects, i.e., correctness, type I, type II, quality achieved, inspection, and waste inspection. As fault location was further refined, the accuracy diminished slightly. This was due to some experimental data that were downloaded from different sources contained unidentified faults not being detected by the model. Consequently, individual metric can be gauged to see how it attributes to faultiness.

7.2 Conclusion

In the domain of software fault prediction, many techniques, i.e., neural networks and fuzzy logic, are applied with enormous amounts of data to construct predictive models. Those models focus on software faultiness without attempting to identify fault type and fault location. This dissertation proposed a systematic fuzzy logic and neural network approach [2, 4, 5] to predicting software fault in software system without test run based on software metrics. This approach can predict fault-proneness and fault type due to inheritance in software system with high prediction accuracy by means of a set of relevant metrics established by the proposed MASP model. To complement the fault prediction, dynamic faults and their corresponding location are determined using a set of graphs and metrics extracted from those graphs. The resulting metrics are subsequently used as a clue for investigating cause of fault.

7.3 Future Work

Besides inheritance, residual faults in software may associate with other object-oriented aspects, such as polymorphism and aggregation, which are beyond the scope of this dissertation. Existing researches concerning these issues are still not extensive enough to be used as a basis for polymorphic and aggregation fault predictions. Consequently, further study of software fault due to polymorphism and aggregation is necessary to create a frame work for predicting those faults.

References

1. Musa, J.D., Iannino, A., and Okumoto, K. *Software Reliability Measurement, Prediction, Application*. New York:McGraw-Hill, (1987).

2. Mahaweerawat, A., Sophatsathit, P., and Lursinsap, C. Software Fault Prediction Using Fuzzy Clustering and Radial-Basis Function Network. *Proceedings of the International Conference on Intelligent Technologies, InTech/VJFuzzy2002* (December 2002): 304-313.

3. Alexander, R. T., Offutt, J., and Bieman, J. M. Syntactic Fault Patterns in OO Programs. *Proceedings of the Eight International Conference on Engineering of Complex Computer Software* (2002):193-202.

4. Mahaweerawat, A., Sophatsathit, P., Lursinsap, C., andMusilek, P. Fault prediction in object-oriented software using neural network techniques. *Proceedings of the International Conference on Intelligent Technologies 2004, InTech'04* (December 2004): 27-34.

5. Mahaweerawat, A., Sophatsathit, P., Lursinsap, C., and Musilek, P. Masp a enhanced model of fault type identification in object-oriented software engineering. *Journal of Advanced Computational Intelligence and Intelligent Informatics* 10, 3 (2006): 312-322

6. Karunanithi, N., Malaiya, Y.K., and Whitley, D. Prediction of software reliability using neural networks. *Proceeding of the International Symposium on Software Reliability Engineering* (1991): 124-130.

7. Lanubile, F. Evaluating Predictive Models Derived from Software Measures. *Journal of Systems and Software* 38, (1996): 225-234

8. Adnan, W. A., Yaakob, M., Anas, R., and Tamjis, M. R. Artificial Neural Network

for Software Reliability Assessment. *TECON 2000 Proceedings* 3, (2000): 446–451

9. Aljahdali, S. H., Sheta, A., and Rine, D. Prediction of Software Reliability: A Comparison between Regression and Neural Network Non-Parametic Models. *ACS/IEEE International Conference on Computer Systems and Applications*, (2001): 470-473.

10. Hochman, R. Software Reliability Engineering: An Evolutionary Neural Network Approach. Master's thesis, Florida Atlantic University, 1997.

11. Avritzer, A. and Weyuker, E. J. Detecting Failed Processes Using Fault Signatures. *IEEE International Proceeding, Computer Performance and Dependability Symposium* (1996): 302-311.

12. Xu, Z. and Allen, E. B. Prediction of Software Faults Using Fuzzy Nonlinear Regression Modelling. *IEEE International Symposium on High Accurance Systems Engineering*,(2000): 281-290.

13. Khoshgoftaar, T. M. and Muson, J. C. Predicting Software Development Errors Using Software Complexity Metrics. *IEEE Journal on Selected Areas in Communications* 8, 2 (1990): 253-261.

14. Khoshgoftaar, T. M. and Allen, E. B. Using Product, Process, and Execution Metrics to Predict Fault-Prone Software Modules with Classification Trees. *Fifth IEEE International Symposium on High Assurance Systems Engineering* (2000): 301-310.

15. Fenton, N.E. and Ohisson, N. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transaction on Software Engineering* 26, 8 (2000): 797-814.

16. Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. Predicting Fault Incidence

Using Software Change History. *IEEE Transactions on Software Engineering* 26, 7 (2000): 653-661.

17. Pedrycz, W., Succi, G., Reformat, M., Musilek, P., and Bai, X. Self Organizing Maps As A Tool For Software Analysis. *Canadian Conference of Electrical and Computer Engineering* (2001): 93-97.

18. Khoshgoftaar, T.M., Pandya, A.S., and More, H.B. A Neural Network Approach For Predicting Software Development Faults. *Proc. Third International Symposium on Software Reliability Engineering* (1992): 83-89.

19. Khoshgoftaar, T. M. and Szabo, R. M. Using Neural Networks to Predict Software Faults During Testing. *IEEE Transaction on Reliability* 45, 3 (1996): 456-462.

20. Yuan, X., Khoshgoftaar, T.M., Allen, E.B., and Ganesant, K. An Application of Fuzzy Clustering to Software Quality Prediction. *Proc. 2000 IEEE Symposium on Application-Specific Systems and Software Engineering Technology* (2000): 85-90.

21. Kamiya, T., Kusumoto, S. and Inoue, K. Prediction of Fault-Proneness at Early Phase in Object-Oriented Development. *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (May 1999): 253-258.

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

23. Kokol, P., Podgorelec, V., Zorman, M., and Šprogar, M. An Analysis of Software Correctness Prediction Methods. *Proceedings of the Second Asia-Pacific Conference on Quality Software (APAQS'01)* (December 2001): 33-39.

24. Thwin, M. M. T. and Quah, T. S. Application of Neural Network for Predicting

Software Development Faults Using Object-Oriented Design Metrics. *Proceedings of the* 9th *International Conference on Neural Information Processing* (November 2002): 2312-2316.

25. Emam, L., Wüst, J., and Daly, J. W. The Prediction of Faulty classes Using Object-Oriented Design Metrics. *Journal of Systems and Software* 56, (2001): 63-75.

26. Briand, L., Wüst, J., and Daly, J. W. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. *Journal of Systems and Software* 51, (2000): 245-273.

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

28. Mao, Y., Sahraoui, H. A., and Lounis, H. Reusability Hypothesis Verification Using Machine Learning Techniques: a case study. *Proceedings of the* 13th *IEEE International Conference on Automated Software Engineering* (1998): 84-93.

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

30. Fioravanti, F. and Nesi, P. A study on fault-proneness detection of Object-Oriented systems. *Proceedings of the fifth Conference on Software Maintenance and Reengineering (CSMR'01)* (March 2001): 121-130.

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

32. Center For High Integrity Software Systems Assurance (Online). Available from: *http://hissa.nist.gov/chissa/ SEI Framework/framework 1.html* [20 January 2002].

33. Fenton, N.E. and Pfleeger, S.L. *Software Metrics: A Rigorous and Practical Approach*. London: International Thomson Computer Press, (1997).

34. Chidamber, S. R. and Kemerer, C. F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (1994): 476-493.

35. Offutt, J. and Alexander, R. A Fault Model for Subtype Inheritance and Polymorphism. 12th *International Symposium on Software Reliability Engineering* (November 2001): 84-95.

36. Wilkie, G. *Object-Oriented Software Engineering*. New York:Addison Wesley, (1993): 14-41.

37. Chen, J. L. and Wang, F. J. Flow analysis of class relationships for object-oriented programs. *Journal of Information Science and Engineering* 16, (2000): 619-647.

38. Guéhéneuc, Y. G. and Albin-Amiot, H. A pragmatic study of binary class relationships. *Proceedings of the 18 th IEEE International Conference on Automated Software Engineering (ASE'03)* (October 2003): 277-280.

39. Kung, D. and Hsia, P. A reverse engineering approach for software testing of objectoriented programs. *Proceedings of the second IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99)* (Marc 1999): 42-49.

40. Zadeh, L.A. Knowledge representation in fuzzy logic. *IEEE Transactions on Knowledge and Data Engineering* 1, 1 (1998): 89-100.

41. Klir, G.J. and Folger, T.A. *Fuzzy Sets, Uncertainty and Information*. New Jersey:Prentice-Hall, (1988).

42. Eklund, P., Kallin, L., and Riissanen, T. Fuzzy Systems, Lecture notes prepared for courses at the Department of Computing Science at Umeå University, Sweden, (February 2000).

43. Chiu, S. Method and Software for Extracting Fuzzy Classification Rules by Subtractive Clustering. *Fuzzy Information Proceeding Society, Biennial Conference of the North American* (1996): 461-465.

44. Chiu, S.L. Fuzzy model identification based on cluster estimation. *Journal of Intelligent and Fuzzy Systems* 2, 3 (1994): 267-278.

45. Prechelt, L. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks* 11, (1998): 761-767.

46. Scientific Toolworks Inc. *Understand for C++*[Computer software], George, Utah, Available from: *http://www.scitools.com* [18 June 2002].

47. Mitra, P., Murthy, C., and Pal, S. K. Unsupervised Feature Selection Using Feature Similarity. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 3 (2002): 301-312.

48. Baase, S. *Computer Algorithms Introduction to Design and Analysis*. the United States of America:Addison-Wesley Publishing Company, (1988).

49. Chen, J. L. andWang, F. J. An inheritance flow model for class hierarchy analysis. *Information Processing Letters* 66, (1998):309-315.

50. Haykin, S. *Neural Networks*. 2 nd ed. New Jersey: Prentice Hall, (1999): 156-312.

51. Yao, J. T. Sensitivity analysis for data mining. *Proceedings of The 22 nd International Conference of The North American Fuzzy Information Processing Society (NAFIPS)* (July 2003): 272-277.

52. Saltelli, A., Chan, K., and Scott, E. M. *Sensitivity Analysis*. the United States of America:John Wiley and Sons, (2000).

53. Lei, L., Wu, N., and Liu, P. Appling sensitivity analysis to missing data in classifiers. *Proceedings of International Conference on Services Systems and Services Management (ICSSSM05)* (June 2005): 1051-1056.

54. Frey, H. C. and Patil, H. C. Identification and review of sensitivity analysis method. *Risk Analysis*, 22, 3 (2002): 553-578.

55. Zurada, J. M., Malinowski, A., and Cloete, I. Sensitivity analysis for minimization of input data dimension for feedforward neural network. *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)* (June 1994): 447-450.

56. Yao, J. t., Teng, N., Poh, H. L., and Tan, C. L. Forecasting and analysis of marketing data using neural networks. *Journal of Information Science AND Engineering* 14, (1998): 843-862.

57. Lu, M. and Yeung, D. S. Application of an artificial neural network based sensitivity analysis technique in concreting productivity study. *Proceedings of the Second International Conference on Machine Learning and Cybernetics, Xi'an* (November 2003): 1207-1212.

58. Lo, J. H., Huang, C. Y., Kuo, S. Y., and Lyu, M. R. Sensitivity analysis of software reliability for component-based software applications. *Proceedings of the 27 th Annual International Computer Software and Applications Conference (COMPSAC'03)* (November 2003): 500-505.

59. Li, M. and Smidts, C. S. A ranking of software engineering measures based on expert opinion. *IEEE Transactions on Software Engineering* 29, 9 (2003): 811-824.

60. Cangussu, J. W., DeCarlo, R. A., and Mathur, A. P. Using sensitivity analysis to validate a state variable model of the software test process. *IEEE Transactions on Software Engineering*, 29, 5 (2003): 430-443.

Vita

Name: Miss Atchara Mahaweerawat.

Date of Birth: 14th October 1976.

Education:

- Ph.D. Program in Computer Science, Department of Mathematics, Chulalongkorn University, Thailand (May 2000 - October 2006).
- Visiting Ph.D. researcher in FACIA at University of Alberta, Alberta, Canada (December 2003 - September 2004).
- B.Sc. in Computer Science (2nd Class Honors), Khonkaen University, Khon Kaen, Thailand (June 1994 - February 1998).

Publication:

- Mahaweerawat, A., Sophatsathit, P., Lursinsap, C., and Musilek, P. “Masp an enhanced model of fault type identification in object-oriented software engineering”. *Journal of Advanced Computational Intelligence and Intelligent Informatics* 10, 3, (2006): 312-322.
- Mahaweerawat, A., Sophatsathit, P., Lursinsap, C., and Musilek, P. “Fault prediction in object-oriented software using neural network techniques”. *Proceedings of the International Conference on Intelligent Technologies 2004, InTech’04, Houston, TX, U.S.A.* (December 2-4, 2004): 27-34.
- Mahaweerawat, A., Sophatsathit, P., and Lursinsap, C. “Software Fault Prediction Using Fuzzy Clustering and Radial-Basis Function Network”. *Proceedings of the International Conference on Intelligent Technologies, InTech/VJFuzzy2002, Hanoi, Vietnam* (December 3-5, 2002): 304-313.

Scholarship: The Office of Higher Education Commission, Ministry of Education, Thailand.