Université de Montréal

Software Stability Assessment Using Multiple Prediction Models

Par

Hong Zhang

Le Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémorire présenté à la Faculté des études supérieures

En vue de l'obtention du grade de

Maîtrise ès sciences (M.Sc.)

En informatique

Juin, 2003

Université de Montréal

Faculté des études supérieures

Ce mémoire de maîtrise intitulé

Software Stability Assessment Using Multiple Prediction Models

Présenté par

Hong Zhang

A été évalué par un jury composé des personnes suivantes :

Présidente rapporteur: EL MABROUK, Nadia

Directeur de recherche : SAHRAOUI, Houari

Membre du jury : ABOULHAMID, El Mostapha

Mémoire accepté le : 31 juillet 2003

Résumé

La qualité de logiciel est actuellement de plus en plus un souci des organizations. La manière la plus populaire d'assurer la qualité de logiciel est d'appliquer des modèles de prévision de qualité de logiciel. Les modèles de prévision peuvent aider dans l'évaluation de beaucoup d'aspects de qualité de logiciel pendant l'étape de développement de logiciel; par exemple, entretien, réutilisabilité, fiabilité et stabilité. En effet, les modèles de prévision deviennent une méthode efficace pour contrôler la qualité de logiciel avant que l'ensemble des progiciels soit déployé, ou pour prévoir la qualité du logiciel avant qu'il soit utilisé. Pendant les dix dernières années, beaucoup d'études liées à ce sujet ont été publiées et un grand nombre de modèles de prévision de qualité ont été proposés dans la littérature. Cependant, établir les modèles de prévision de qualité de logiciel est une tâche complexe et à ressources consumantes.

En général, il y a deux approches de base pour construire les modèles de prévision de qualité de logiciel. La première établit automatiquement le modèle avec des données historiques. La seconde fait participer des experts établissant le modèle manuellement.

La première approche se fonde sur des données de mesure historiques pour accomplir son but. La qualité de ces modèles dépend fortement de la qualité des échantillons utilisés. Malheureusement, la qualité des échantillons disponibles est habituellement pauvre en programmation. La quantité limitée de données disponibles pour ces modèles le rend difficile à généraliser, valider, et de réutiliser les modèles existants. En effet, contrairement à d'autres domaines, les petites tailles et l'hétérogénéité des échantillons de rendent très difficile de dériver des modèles largement applicables.

La connaissance extraite de l'heuristique domaine-spécifique est employée par la deuxième approche pour établir les modèles de prévision de qualité de logiciel. Les modèles obtenus emploient des jugements des experts, et vise à établir un rapport

intuitivement acceptable entre les attributs internes de logiciel et une caractéristique de qualité. Bien que ces modèles soient adaptés au processus décisionnaire, il est difficile de les généraliser faute de connaissance commune et largement admise dans le domaine de qualité de logiciel.

A cause du manque de données historiques ou de connaissance experte dans un domaine spécifique, il est difficile d'établir systématiquement les modèles de prévision spécifiques. Une alternative est de choisir un modèle de prévision existant. Mais les modèles spécifiques obtenus à partir d'une situation particulière ne sont pas assez généraux pour être efficacement applicables. Par conséquent, le choix d'un modèle approprié est une décision difficile et non triviale pour une compagnie.

Dans notre thèse, nous proposons une approche de combinaison pour résoudre ce problème. L'idée principale est de combiner et adapter les modèles existants de telle manière que le modèle combiné fonctionne bien sur un système particulier ou dans un type d'organisation particulier. En outre, nous visons également à améliorer les capacités de prévision des modèles existants.

L'approche de combinaison est recommandée comme une manière efficace pour améliorer les modèles de simple-issue utilisés actuellement. Nous employons un algorithme génétique pour mettre en application notre approche de combinaison. Dans notre solution proposée, nous supposons que les modèles de prévision existants sont l'arbre de décision ou les classificateurs basés sur les règles. Les résultats d'essai indiquent que l'approche de combinaison proposée avec un algorithme génétique peut améliorer les capacités de prévision des modèles existants de manière significative dans un contexte de systèmes multiple.

Mote clés : Modèle de Prévision de Qualité de Logiciel, Métrique Logiciel, Algorithme Génétique

Abstract

Software quality is a concern of more and more organizations now. The most popular way to assure software quality at present is to apply software quality prediction models. Prediction models can help in the evaluation of many aspects of software quality during the software development stage; such as, maintainability, reusability, reliability and stability. In fact, prediction models are becoming an efficient way to control software quality before software packages are deployed, or to predict the quality of the software before they are used. During the past ten years, a lot of studies related to this subject have been published and a large number of quality prediction models have been proposed in the literature. However, building software quality prediction models is a complex and resource-consuming task.

In general, there are two basic approaches to building software quality prediction models. The first one uses historical data to build the model automatically. The second one involves experts building the model manually.

The first approach relies on historical measurement data to accomplish its goal. The quality of these models depends heavily on the quality of the samples used. Unfortunately, the quality of samples available is usually poor in software engineering. The limited amount of data available for these models makes it difficult to generalize, to cross-validate, and to reuse existing models. Indeed, contrary to other domains, the small sizes and the heterogeneity of the samples makes it very difficult to derive widely applicable models.

Knowledge extracted from domain-specific heuristics is used by the second approach to build software quality prediction models. The obtained prediction models use judgments from experts, and aim to establish an intuitively acceptable causal relationship between internal software attributes and a quality characteristic. Although

these models are adapted to the thought decision-making process, they are also hard to generalize because of a lack of widely accepted common knowledge in the field of software quality.

Due to the lack of historical data or the lack of expert knowledge in a specific domain, it is hard to build organizationally specific prediction models. An alternative can be to choose an existing prediction model. But the specific models obtained from a particular situation are not general enough to be efficiently applicable. As a consequence, selecting an appropriate quality model is a difficult and non-trivial decision for a company.

In our thesis, we propose a combination approach to solve this problem. The main idea is to combine and adapt existing models in such a way that the combined model works well on a particular system or in a particular type of organization. In addition we also aim at improving the prediction ability of existing models.

The combination approach is recommended as an efficient way to improve on the single-issue models used at present. We use a genetic algorithm to implement our combination approach. For our proposed solution, we assume that the existing prediction models are the decision tree or rule-based classifiers. The test results indicate that the proposed combination approach with a genetic algorithm can significantly improve the prediction ability of existing models within a multiple systems context.

Key words: Software Quality Prediction Model, Software Metric, Genetic Algorithm,

CONTENTS

Chapter 4 Combination Algorithm

Chapter 5 Implementation and Experimentation

Chapter 6 Conclusion

LIST OF FIGURES

LIST OF TABLES

Acknowledgements

I would like to express my sincere appreciation to my director, Dr. Houari Sahraoui for his invaluable guidance, his encouragement, his care and support throughout the course of this thesis work. I really appreciate the opportunity to work on such an interesting project with his guidance and advice. He has constantly supported me in a very kind and encouraging way by pointing out relevant research and generating interesting ideas.

I also would like to make a special acknowledgement to Mr. Salah Bouktif for his collaboration in the development and implementation of the Algorithms. I benefited greatly from formal and informal discussions with him. I am also thankful to Mr. Mohammed Rouatbi, who offered me his efficient fitness function used by the genetic algorithm.

I also truly thank my brother Zhang Yi Wei. This thesis has benefited from his careful reading and constructive criticism.

I also express my thanks to my friends Chen Ji Ling, Qin Li Sheng, Shen Shi Qiang, Mai Gang, Zheng Suo Shi and Wu Lei for their help, friendship and encouragement.

Finally, I wish to express a special acknowledgement to my wife Jia Dong Mei, my daughter Yue Ran Zhang and my son Gary Zhang for their support, understanding, and for making all of this possible.

Chapter 1 Introduction

1.1 Motivation

Computer use is now prevalent in almost all aspects of our everyday life; consequently, software has become critical to the development and maintenance of consumer products. Now, more than ever, software developers are concerned with software quality when developing new products. The reason for this stems from the immense demands on people, money and time when developing software products because software is becoming larger and more complex [35]. The question of how to develop high-quality software is critical. One of the best ways to assure software quality is to address this issue and make accurate predictions before the software is developed.

Predicting software quality is a complex and resource-consuming task. The process of predicting defects in the early stage of the software lifecycle has become a major undertaking for software engineers. Over the last 30 years a great deal of research has been undertaken in an attempt to predict software quality [24]. There are many papers advocating statistical models and metrics which purport to answer the quality question. Many of the studies related to this issue have added to our knowledge base. For example, numerous software metrics have been developed and some of the prediction models built from these metrics have been found to be effective tools in controlling the software quality. In fact, prediction models are becoming an efficient way to control software quality both before and during software development.

Recently, many prediction models have been proposed to predict certain aspects of software quality: such as, maintainability, reusability, reliability, and stability to name a few [24]. Most of these prediction models have been built using statistical methods, which require historical data. Unfortunately, many software developers lack historical data. Therefore, it is hard to build organizationally specific prediction models because of this lack of information. Consequently, the alternative has been to choose existing prediction models. However existing models are specific to a particular situation and consequently are not general enough to be efficiently applied to other contexts. More significantly, many prediction models tend to model only part of the underlying problem within a context: therefore, they are not universal. Intuitively, a way to solve this problem might be by collecting data from different kinds of application contexts to build universal software quality prediction models. Unfortunately, this would be too complex and too time consuming to achieve. Furthermore, in practice, it's almost impossible to obtain all the data necessary for prediction models to be built. Therefore, in our research, we hope to address these problems by proposing a combination algorithm to obtain a cross-valid software quality prediction model.

1.2 Goals

In general, software quality prediction models are obtained from historical measurement data or the domain specific heuristics of experts. The main purpose of this research is to find another approach to establishing quality prediction models through the combination of existing models in order to obtain new modes. This new prediction model is neither from the historical measurement data nor from experts' knowledge.

The combined models obtained from existing prediction models can be an alternative for software organizations that lack historical data. This approach is achievable because many prediction models, which can only satisfactorily work for the specific circumstances from which they were built, have been proposed in the last few decades.

We propose an approach that combines these existing prediction models from various contexts, using a genetic algorithm, with the goal being to produce a more universally applicable "combining model" for software stability prediction. The results obtained from this study show that this approach is more efficient and the prediction accuracy is higher in the testing data sets.

Therefore, not only has it become effective and efficient to produce a predictive model from the existing predictive models of these organizations. Their results can lead to higher prediction accuracy rate that requires less effort during the implementation phase and makes the design phase more efficient.

The "combining model" approach is illustrated in Figure 1.1. Each cylinder indicates the source data set from which the prediction models are built. Each rectangle indicates the approach to building the prediction models. Each ellipse indicates the prediction model. In the literature, only the prediction models and some of the approaches are presented. That is, the source data sets are unknown. This research focuses on the stability rule-based prediction models. Our aim is to find an approach to build a new model from the posted models using a genetic algorithm. It is hoped that both this approach and this new model can be applied widely. Using this new approach, it is not necessary to have the original source data sets because our input data actually is a set of existing prediction models. Our outputs are some new prediction models that are more general and more accurate in estimating software quality. Moreover, this study is also an exploratory phase that offers proof to the concept of combining existing models with the genetic algorithm. Some techniques and results of this study have been improved by the students that continue the research [55].

Therefore, the goal of this study is to propose and verify the 'combining' approach, by using a genetic algorithm, as an efficient method to develop cross-valid software quality prediction models.

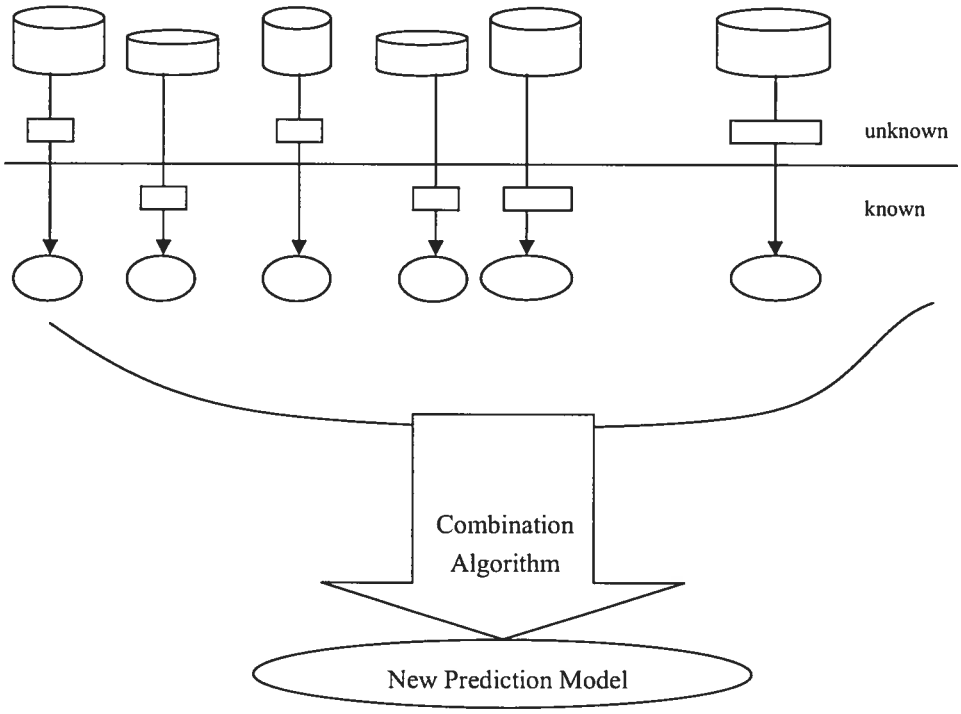


Figure 1.1 The Approach and Concept of Our Research

1.3 Contributions

Our combination algorithm was validated in a "semi-real" environment. In this evaluation environment, all data collected was from some real software systems and the models were extracted from part of this data set through the C4.5 algorithm [51]. We trained the original models and tested the combination models with a10-fold cross validation technique in this real data environment also. The results coming from our experiment are: that a certain kind of local search method, such as a genetic algorithm, can be used as an evolutionary approach for combining and improving software quality prediction models in a particular context.

Our research contribution focuses on the following two aspects:

First, we propose an approach to using a genetic algorithm for the improvement of prediction models through their combination.

Second, we show that this approach can work well for the classes interface stability prediction in real software systems.

1.4 Outline

In this study, we apply a genetic algorithm (GA) as a combination approach to build more satisfactory software quality prediction models and optimize the prediction accuracy of the new models.

In Chapter 2, we present a review of the concepts of software quality and its prediction models, as well as a description of some software quality prediction models that have been posted in the literature in order to provide an example of other prediction models. In Chapter 3, we describe the GA principles in general, such as GA operators and parameters. The research methodology of our combination algorithm is presented in Chapter 4, while Chapter 5, describes the implementation of our experiment using the algorithm on the stability prediction models.

Finally, in Chapter 6, conclusions are made and a brief summary is presented. Furthermore, problems concerning optimizing quality estimation models with this specific technique and future work are presented.

Chapter 2 Software Quality Prediction Models

With the expanding application of computers in many aspects of our lives, the use of computer software has also become a necessary part of our everyday life. Like computer hardware, computer software is a consumer product as well. With increasing competition in the software market, software quality is a key concern for the software vender/producer because the market will only accept the best quality products. Similarly, software quality is now of greater concern to computer users, because to most users, the investment in software is a long term one and it usually directly affects the efficiency of their computer operations. Therefore, developers have had to address this issue in order to maintain consumer satisfaction.

Aside from consumer demand, the concern for software quality is a central and critical issue for software companies because the development of software requires immense amounts of time, money and human resources to produce. Therefore, it is necessary for companies to eliminate or reduce software defects in order for their product and their company to survive.

In this chapter we give a brief overview of the concepts related to software quality and its prediction models.

2.1 Terminology of Software Quality

Before discussing software quality, it is necessary to consider the definition of a **software product**. A widely accepted definition is that: *a software work product is any artifact created as part of the software process including computer programs, plans procedures, and associated documentation and data* [50].

From this definition, the term "software quality" can be applied to both the product being produced and the process used by software engineers to produce it. Therefore, there are two types of quality, **product quality** and **process quality**. Although they are dependent on each other, they involve different techniques and measures, and have different implications. Product quality is easy to understand, but the term process quality is not that intuitively simple. Therefore, we need to clarify what is a **software process**. A software process is a set of activities, methods, practices, and transformations that people use to develop and maintain software work products [50]. Now we can look at the contents of the two types of software quality.

- **Product quality**

  Broadly speaking, product quality is related to how well the product satisfies its customers' requirements. Related to this are the usability, performance, reliability, and the maintainability of the software [30].

- **Process quality**

  This is concerned with how well the process used to develop the product worked. Usually researchers are concerned with elements such as cost estimation and schedule accuracy, productivity, and the effectiveness of various quality control techniques [30].

From the above descriptions, we can see that the definition of software quality in literature contains many aspects. In our study, we do not want to take up too much space on the various detailed aspects. Instead, we adopt a simple but clear definition:

- **Software quality**

  Software quality can be thought of as the number and frequency of problems and

defects discovered [50].

The most important terms associated with this definition of software quality are **software defects** and **software problems**. The following definitions will clarify these two terms.

- **Software Defects**

  A software defect is any flaw or imperfection in a software work product or software process [50].

It is any unintended characteristic that impairs the utility or worth of an item, or any kind of shortcoming, imperfection, or deficiency. A software defect is a manifestation of a human (software producer) mistake. However, not all human mistakes are defects, nor are all defects the result of human mistakes. When found in executable code, a defect is frequently referred to as a fault or a bug. A fault is an incorrect program step, process, or data definition in a computer program. Faults are defects that have persisted in software until the software is executable.

Software defects include all defects that have been encountered or discovered by examination or operation of the software product. Possible values in this subtype are as follows:

- Requirement defect
- Design defect
- Code defect
- Document defect
- Test case defect
- Other work product defect

- **Software Problems**

  Software problems are another quality concern related to software products. A software problem is a human encounter with software that causes difficulty, doubt,

or uncertainty in the use or examination of the software [30].

A software problem has typically been associated with that of a customer identifying, in some way, a malfunction in the program. The notion of a software problem is beyond that of an unhappy customer. There are many terms used for problem reports throughout the software community; for example, incident reports, customer service requests, trouble reports, inspection reports, error reports, defect reports, failure reports and test incidents. In a generic sense, they all stem from a person's unsatisfactory encounter with the software. Software problems are human events. The encounter may be with an operational system (dynamic), or it may be an encounter with a program listing (a static encounter.)

In a dynamic (operational) environment, some problems may be caused by failures. According to Musa in "*Software Reliability Measurement, Prediction, Application*", a **failure** is the departure of software operations from requirements. A software failure must occur during the execution of a program. Software failures are caused by faults, that is, defects found in executable code [47].

In a static (non-operational) environment, such as a code inspection, some problems may be caused by defects. In both dynamic and static environments, problems also may be caused by misunderstandings, misuse, or a number of other factors that are not related to the software product being used or examined.

2.2 Software Quality Assurance

Software Quality Assurance (SQA) is the main approach used to provide good quality software. There has been remarkable progress made in SQA since the early days of computing. At the beginning, the process of developing software products was simply about writing procedures to perform given tasks. The most common and popular way of assuring the quality of software was through program testing. This means that software

quality was treated as an afterthought or as a postscript in software development. Hilburn and Towhidnejad argued that software quality should be addressed in the front-end of the lifecycle and should not be ignored until after the development of the product [35]. They suggested that quality should be focused on during the whole software development process. Figure 2.1 developed by Hilburn and Towhidnejad, shows a V Quality model that provides a conceptual framework for such a focus.

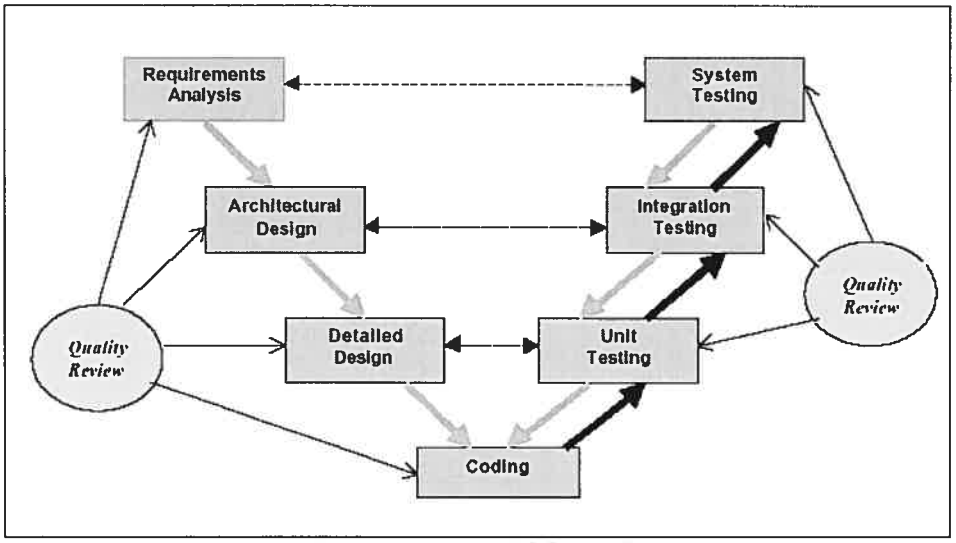


Figure 2.1 The V Model for Quality

During the test phase, only the functional requirement can be determined. Aside from the functional requirement, there are other requirements; such as, maintainability, reusability, reliability, and stability that need to be determined. Unfortunately, these cannot be determined through testing. As a consequence of this problem, software quality has been treated as an afterthought in the software development process. This solution does not appear to adequately address the quality issue; therefore, a better possible solution may be to apply **software prediction models** to assure software quality during the development lifecycle [53].

Software prediction models address the evaluation of software quality during the software development life cycle. The prediction model, specified for a specific project, consists of a set of important quality characteristics. In general there are six

characteristics of software that can be used as criteria for quality as defined in ISO/IEC 9126 (See Table 2.1).

Table 2.1 Software Characteristics from ISO/IEC 9216

Characteristic	Explanation
Functionality	Attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy a stated of implied need.
Reliability	Attributes, that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
Usability	Attributes that bear on the effort needed for use, and on the individual evaluation of such use, by a stated or implied set of users.
Efficiency	Attributes that bear on the relationship between the level of the performance of the software and the amount of resources used, under stated conditions.
Maintainability	Attributes that bear on the effort needed to make specified modifications.
Portability	Attributes that bear on the ability of software to be transformed from one environment to another.

2.3 Software Measurement and Metrics

Software measurement is another important concept that is concerned with deriving numeric values for some attributes of a software product or a software process. These values enable people to intuitively evaluate and draw conclusions about the quality of the software or the software process. Some large companies have introduced program metrics for measurement purposes and are using collected metrics in their quality management processes [56]. Most of the focus has been on collecting metrics on the program and the processes of verification and validation. During the past decades, a lot of people (such as Offen, Jeffrey, Hall, and Fenton) have contributed for the introduction of software metrics as a way to improve software quality.

A software metric is any type of measurement that relates to a software system, process or related documentation [56]. For example, lines of code are the measurement of the size of a software product. The Fog index (Gunning, 1962) is a measure of the readability of a passage of written text. The number of reported faults in a delivered software product or the number of person-days required to develop a system component are also example of software metrics.

- **Control Metrics and Predictor Metrics**

There are two types of software metrics to consider: **control metrics** and **predictor metrics**. Control metrics are usually associated with software processes (therefore they are also called process metrics by some researchers) while predictor metrics are associated with software products. Examples of control (or process) metrics are the average effort and time required to repair reported defects. Examples of predictor metrics include the cyclomatic complexity of a module, the average length of an identifier in a program, or the number of attributes and operations associated with objects in a design. Both control and predictor metrics may influence management decision making as shown in Figure 2.2.

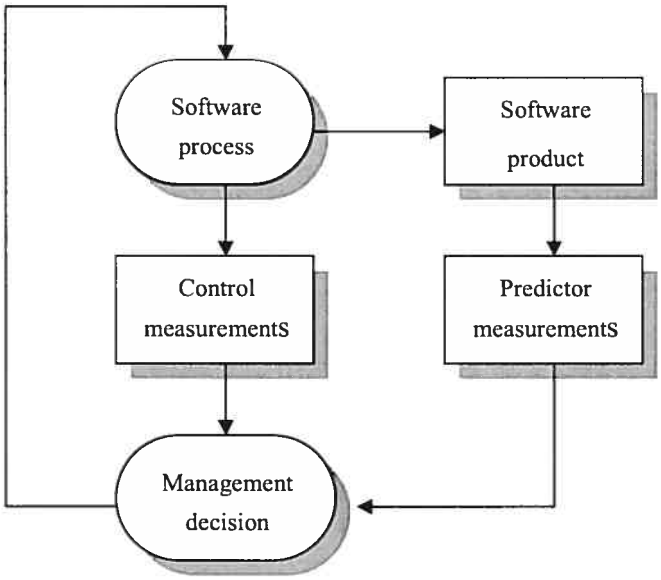


Figure 2.2 Predictor and Control Metrics

- **Dynamic Metrics and Static Metrics**

Predictor metrics are concerned with characteristic of the software itself. Unfortunately, software characteristics, such as size and cyclomatic complexity that can be easily measured, do not have a clear and universal relationship with quality attributes such as understandability and maintainability. The relationships vary depending on the development process, technology and the type of system being developed. Organizations that are interested in software measurements have to construct a historical database, which can be used to discover how the software product attributes are related to the qualities of interest in the organization.

Product metrics fall into two classes:

1. Dynamic metrics, which are the collected measurements made of a program in execution.
2. Static metrics, which are the collected measurements made of the system representations such as the design, program or documentation.

The two different types of metrics are related to different quality attributes. Dynamic metrics are to evaluate the efficiency and the reliability of a program whereas static metrics are to evaluate the complexity, understandability and maintainability of a software system.

Dynamic metrics are usually directly related to software quality attributes. They are relatively easy to measure. For example, the execution time required for particular functions and the time required to startup a system are dynamic metrics. These relate metrics directly to the system's efficiency.

Static metrics, on the other hand, have an indirect relationship to quality attributes. There are a large number of these metrics proposed and experiments conducted to derive and validate the relationships between these metrics and system complexity, understandability and maintainability. Table 2.2 lists several static metrics used for assessing quality attributes. Among these, program/component length and control

complexity seem to be the most reliable predictors of system understandability, complexity and maintainability [56].

All of the metrics in Table 2.2 are for function-oriented designs. Their usefulness as predictor metrics is still being established despite the increasing popularity of object-oriented software systems.

Table 2.2 Function-Oriented Software Product Metrics

Software Metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions that call some other function (say X). Fan-out is the number of the functions which are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and the changes to X will have extensive knock-on effects. A high value for fan-out suggests that overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a program component, the more complex and error-prone that component is likely to be.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of Conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document may be to understand.

- **Object-oriented Metrics**

Since the early 1990s, there have been a number of studies concerning object-oriented metrics. Some of these were derived from the previously existing metrics shown in Table 2.2, but others are unique to object-oriented systems. Table 2.3 explains some of the object-oriented metrics.

These specific metrics are depending on the project itself, the goals of the quality management team and the type of software developed. In some situations, all the

metrics in Table 2.2.and Table 2.3 may be useful. However, there are situations where some metrics are inappropriate. Organizations should choose the most appropriate metrics for their needs.

Table 2.3 Object-oriented Metrics

Object-oriented Metric	Description
Depth of inheritance tree	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design as, potentially, many different object classes have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described in Table2.2 and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external method.
Weighted methods per class	This is the number of methods included in a class weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as superclasses in and inheritance tree.
Number of overriding operations	These are the number of operations in a superclass which are overridden in a subclass. A high value for this metric indicates that the superclass used may not be an appropriate parent for subclass

- **Relations between Internal and External Attributes**

Software quality characteristics are also categorized as **internal** or **external** by some researchers. The size, inheritance, and coupling are internal attributes and can be directly measured. While the external characteristics of maintainability, reusability, and reliability can only be measured after a certain time of use. In order to predict software quality characteristics, software attributes (or metrics) were introduced because their properties are directly measurable. Roughly speaking, building a software quality prediction model is akin to building a relationship between the measurable internal attributes and the external characteristics. Therefore, before talking about software quality prediction models, we also need to consider the measurable attributes of software and the software measurements which are introduced in the following.

Some software quality attributes (mostly the external attributes) are impossible to measure directly. Attributes such as maintainability, complexity and understandability are affected by many different factors. There are no straightforward metrics for them. Therefore we have to measure some internal attribute of the software (such as its size) with the assumption that there is a relationship between what we can measure and what we want to know. Ideally, there should be a validated and clear relationship between the software external and internal attributes.

Figure 2.3 shows some external quality attributes that might be of interest [56]. On the diagram's left side are some external attributes and on the right side are some internal ones. This diagram shows that the measurable internal attributes might be related to the external attributes. It suggests that there may be a relationship between external and internal attributes but does not say what the relations are.

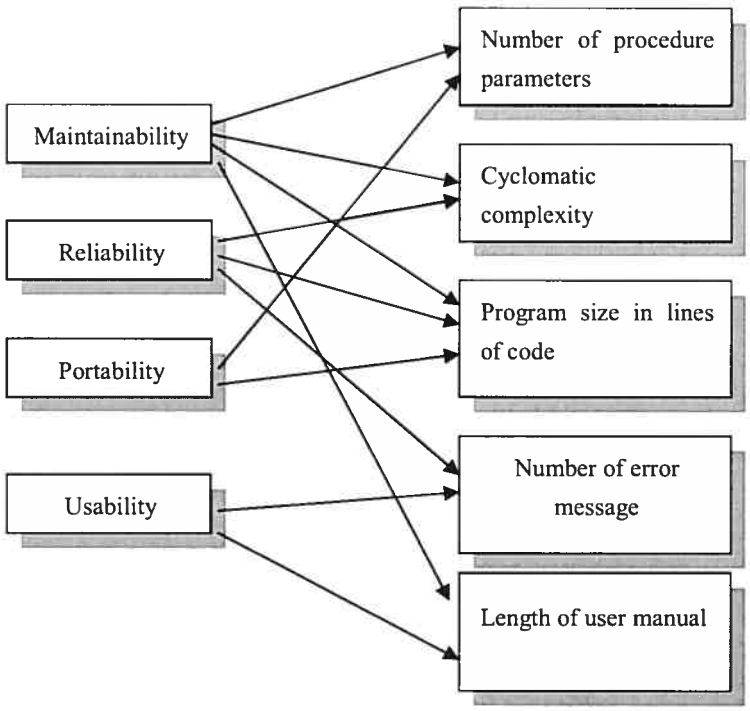


Figure 2.3 Relationships between Internal and External Software Attributes

If a measurement of an internal software attribute is to be a useful predictor of an external one, three conditions must hold (Kithchenham, 1990):

1. The internal attribute must be measured accurately.
2. A relationship must exist between the measurable internal attribute and the external behavioral attribute.
3. This relationship is validated and can be expressed in terms of an understandable formula or model.

The model formulation involves identifying the functional form of the model (i.e. linear, exponential) by analyzing collected data and identifying the parameters which are to be included in the model. Such model development usually requires significant experience in statistical techniques if it is to be trusted. A professional statistician should usually be involved in the process.

The software quality prediction models used in our study are based on the basic elements of a software measurement environment and the metrics described above. We choose 22 structural software metrics to predict its stability. The metrics (see Table 2.4) are grouped in four categories by coupling, cohesion, inheritance, and complexity. They constitute a union of metrics used in different theoretical models [17, 7, 59, 12].

After the software metrics are defined and collected, they can be used to build the relationship between the immeasurable software qualities and the measurable software metrics. The assumed relations are called software quality prediction models.

Table 2.4 The 22 Software Metrics Used as Attributes in Our Experiments

Metrics		Description
Cohesion metrics		
1	LCOM	lack of cohesion methods
2	COH	cohesion
3	COM	cohesion metric
4	COMI	cohesion metric inverse
Coupling metrics		
5	OCMAIC	other class method attribute import coupling
6	OCMAEC	other class method attribute export coupling
7	CUB	number of classes used by a class
8	CUBF	number of classes used by a memb. funct.
Inheritance metrics		
9	NOC	number of children
10	NOP	number of parents
11	NON	number of nested classes
12	NOCONT	number of containing classes
13	DIT	depth of inheritance
14	MDS	message domain size
15	CHM	class hierarchy metric
Size complexity metrics		
16	NOM	number of methods
17	WMC	weighted methods per class
18	WMCLOC	LOC weighted methods per class
19	MCC	McCabe's complexity weighted meth. per cl.
20	DEPCC	operation access metric
21	NPPM	number of public and protected meth. in a cl.
22	NPA	number of public attributes

2.4 Software Quality Prediction Models and Building Approach

As mentioned before, software quality is evaluated in terms of maintainability, reusability, reliability, stability, etc. The majority of these quality characteristics are not directly measurable. But we can use software metric values to help us estimate the software quality. To do this, we have to assume a relationship between them. This is the software quality prediction (estimation) model.

Software prediction models address the evaluation of software quality during the software development life cycle. The prediction model, specified for a specific project, consists of a set of important quality characteristics. These attributes (or metrics) are directly measurable software properties that qualify quality characteristics.

Software quality prediction models offer an interesting solution to assure software quality because they can be used to incorporate a wide variety of quality assurance techniques [53]. Most importantly, software quality prediction models can be used to predict the number of the defects (faults) in software systems before they are deployed [24].

The approach to building software quality prediction models is very complex and source costing. Roughly speaking, building a quality prediction model consists of building a relationship between the internal and external quality characteristics. There are a lot of typical approaches to prediction models; such as, statistic, machine learning, neural networking and BBN.

The work done so far to build efficient and usable software quality prediction models falls into two families. The first one relies on historical measurement data to achieve its goal (see for example [3], [14] and [43]). The quality of these models depends heavily on the quality of the samples used, which is usually poor in software engineering. Indeed, contrary to other domains, the small sizes and the heterogeneity of the samples makes it difficult to derive widely applicable models. As a result, the models may capture trends, but do so by using sample-dependent threshold values [54]. Also, as stated by Fenton & Neil [26], the majority of the produced models are naïve; they cannot serve as decision support during the software development process. This is because often the predictive variables and the quality characteristics used for prediction show no obvious causal link that could explain their derived relationship. The models

behave as simple black boxes that take the predictive variables as input and the predicted variables as output [53].

The second way of building software quality estimation models uses knowledge extracted from domain-specific heuristics. The obtained predictive models use judgments from experts to establish an intuitively acceptable causal relationship between internal software attributes and a quality characteristic. Although they are adapted to the thought decision-making process, these models are hard to generalize because of a lack of widely accepted common knowledge in the field of software quality.

Consequently, there exists a need for an approach that combines the advantages of using both historical measurement data and domain knowledge.

2.5 Existing Software Quality Prediction Models

In fact, prediction models are becoming an efficient way to predict the quality of the software at early stages of development. During the past decades, there have been a lot of studies and papers generated on this topic. Consequently, a large number of proposed quality models have been proposed in the literature. There are many kinds of software quality prediction models. In this section we give an overview of four kinds of prediction models, which fit in one of the following categories:

- Static Regression Models
- Bayesian Belief Networks Models
- Neural Network Models
- Decision Tree Models

2.5.1 Static Regression Software Defect Prediction Models

Most prediction models are based on size and complexity metrics. The earliest such models are typical of many regression based "data fitting" models which became common place in the literature. The results from regression methods showed that linear models of certain simple metrics provide reasonable estimates for the total number of defects D (the dependent variable is actually defined as the sum of the defects found during testing and the defects found during the two months after release). The following represents some regression equations posted in the literature:

$$D = 4.86 + 0.018L \quad \ldots\ldots\ldots\ldots (1)$$

$$D = \frac{V}{3{,}000} \quad \ldots\ldots\ldots\ldots\ldots\ldots (2)$$

$$\frac{D}{L} = A_0 + A_1 \ln L + A_2 \ln^2 L \quad \ldots\ldots (3)$$

$$D = 4.2 + 0.0015(L)^{4/3} \quad \ldots\ldots\ldots (4)$$

The first Equation (1) computed by Akiyama [2], which was based on a system developed at Fujitsu in Japan, predicted defects from lines of code (LOC). From (1) it can be calculated that a 1,000L (it is 1000 LOC) module is expected to have approximately 23 defects.

The second Equation (2) provided by Halstead [34] is a notable equation. This regression model predicts D, the number of defects, depends on a program P. In this equation, V is the (language dependent) volume metric (which like all the Halstead metrics is defined in terms of the number of unique operators and unique operands in P; for details see [23]). The divisor 3,000 represents the mean number of mental discriminations between decisions made by the programmer.

Equation (3) was created by Lipow [41]. In this equation, He got around the problem of

computing V directly by using lines of executable code L instead. Specifically, he used the Halstead theory to compute a series of equations. In equation (3), each of the Ai is dependent on the average number of usages of operators and operands per LOC for a particular language. For example, for Fortran $A_0 = 0.0047;\ A_1 = 0.0023;\ A_2 = 0.000043$.

For an assembly language n $A_0 = 0.0012;\ A_1 = 0.0001;\ A_2 = 0.000002$.

Gaffney [31], argued that the relationship between D and L was not language dependent. In Equation (4), he used Lipow's own data to deduce this prediction model. An interesting ramification of this was that there was an optimal size for individual modules with respect to defect density. For (4) this optimum module size is 877 LOC.

Numerous other researchers have since reported on optimal module sizes. For example, Compton and Withrow of UNISYS derived the following polynomial equation, [19]:

$$D = 0.069 + 0.00156L + 0.00000047(L)^2 \qquad (5)$$

Based on (5) and further analysis Compton and Withrow concluded that the optimum size for an Ada module, with respect to minimizing error density, is 83 source statements.

The realization that size-based metrics alone are poor general predictors of defect density spurred on much research into more discriminating complexity metrics. McCabe's cyclomatic complexity, [45], has been used in many studies, but it too is essentially a size measure (being equal to the number of decisions plus one in most programs). Kitchenham et al. [40], examined the relationship between the changes experienced by two subsystems and a number of metrics, including McCabe's metric. Two different regression equations resulted in (6) and (7):

$$C = 0.042MCI - 0.075N + 0.00001HE \qquad (6)$$

$$C = 0.25MCI - 0.53DI + 0.09VG \qquad (7)$$

For the first subsystem changes, C, was found to be reasonably dependent on machine code instructions, MCI, operator and operand totals, N, and Halstead's effort metric,

*H*E. For the other subsystem McCabe's complexity metric, *VG* was found to partially explain *C* along with machine code instructions, *MCI* and data items, *D*I.

All of the metrics discussed so far are defined in terms of code. There are now a large number of metrics available earlier in the lifecycle of software, most of which have been claimed by their proponents to have some predictive power with respect to residual defect density. For example, there have been numerous attempts to define metrics which can be extracted from design documents using counts of "between module complexity" such as call statements and data flows; the most well known are the metrics in [49]. Ohlsson and Alberg, [4], reported on a study at Ericsson where metrics derived automatically from design documents were used to predict, in particular, fault-prone modules prior to testing. Recently, there have been several attempts, such as [17] and [19], to define metrics on object-oriented designs.

For the regression software defect prediction models, the essential problem is the oversimplification. Typically, the method is for a simple relationship between some predictor and the number of defects delivered. Size or complexity measures are often used as such predictors as mentioned above. The result is a naïve model.

Indeed, such models fail to include all the causal or explanatory variables needed to make the models generalizable. And they can only be used to explain a data set obtained in a specific context. In order to establish a causal relationship between two variables, Bayesian Belief Networks (BBN) was developed to improve the explanatory power.

2.5.2 Bayesian Belief Networks Models

The relationships between product, process attributes and numbers of defects may be too complex to apply straightforward curve fitting models. In predicting defects discovered in a particular project, additional variables can be added to the model, for

example, the number of defects discovered may depend on the effectiveness of the method with which the software is tested. It may also be dependent on the level of detail of the specifications from which the test cases are derived, the care with which requirements have been managed during product development, and so on. The BBN models are the better candidates for situations with such a rich causal structure.

A Bayesian Belief Network (BBN) is a special type of diagram (called a graph) together with an associated set of probability tables. The graph is made up of nodes and arcs where the nodes represent uncertain variables and the arcs the causal/relevance relationships between the variables.

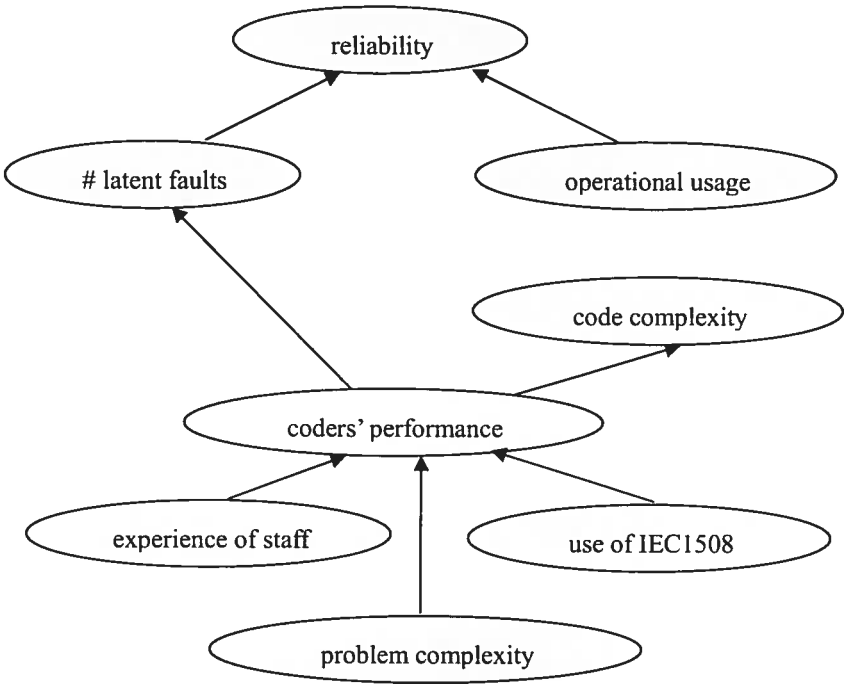
BBN model (also known as graphical probability models) use the subjective judgments of experienced project managers to build the probability model. It can be used to produce forecasts about the software quality throughout the development life cycle. Moreover, the causal or influence structure of the model more naturally mirrors the real world sequence of events and relations that can be achieved with other formalisms.

The relationship between the attributes and the number of defects are too complex that additional variables, such as probability, have to be added to the model. Probability is a dynamic theory. It provides a mechanism for coherently revising the probabilities of events as evidence becomes available [28].

Fenton proposed a BBN model (see Figure 2.4) for an example "reliability prediction" problem in 1999[24]. We take his model and explanation to show the general information of the BBN model.

In Figure 2.4, the nodes represent discrete or continuous variables, for example, the node "use of IEC 1508" (the standard) is discrete having two values "yes" and "no," whereas the node "reliability" might be continuous (such as the probability of failure). The arcs represent causal/influential relationships between variables. For example,

software reliability is defined by the number of (latent) faults and the operational usage (frequency with which faults may be triggered). Hence, this relationship was modeled by drawing arcs from the nodes "number of latent faults" and "operational usage" to "reliability."



NODE PROBABILITY TABLE (NPT) FOR THE NODE "RELIABILITY"

operational usage		low			med			high		
faults		low	med	high	low	med	high	low	med	high
reliability	low	0.10	0.20	0.33	0.20	0.33	0.50	0.20	0.33	0.70
	med	0.20	0.30	0.33	0.30	0.33	0.30	0.30	0.33	0.20
	high	0.70	0.50	0.33	0.50	0.33	0.20	0.50	0.33	0.10

Figure 2.4 "Reliability Prediction" BNN Example

For the node "reliability" the node probability table (NPT) might, therefore, look like that shown in the Figure 2.4 (for ultra-simplicity we have made all nodes discrete so that

here reliability takes on just three discrete values low, medium, and high). The NPTs capture the conditional probabilities of a node given the state of its parent nodes. For nodes without parents (such as "use of IEC 1508" in Figure 3.4) the NPTs are simply the marginal probabilities.

There may be several ways of determining the probabilities for the NPTs. One of the benefits of BBNs stems from the fact that we are able to accommodate both subjective probabilities (elicited from domain experts) and probabilities based on objective data. Recent tool developments mean that it is now possible to build very large BBNs with very large probability tables (including continuous node variables).

The most important advantages of using BBNs is the ability to represent and manipulate complex models that might never be implemented using conventional methods. Another advantage is that the model can predict events based on partial or uncertain data. Because BBNs have a rigorous, mathematical meaning there are software tools that can interpret them and perform the complex calculations needed in their use.

2.5.3 Neural Network Models

In the last decade, significant effort has been put into the research of developing prediction models using neural networks. Many researchers [Khoshgoftaar, 1995] realized the deficiencies of regression methods (see section 2.5.1) and explored neural networks as an alternative. Neural networks are based on the principle of learning from example and no prior information is specified (unlike the Bayesian approach discussed in previous section). Neural networks are characterized in terms of three entities: the neurons, the interconnection structure and the learning algorithm [Karunanithi, 1992].

Neural networks are learning-oriented techniques, which use prior and current knowledge to develop a software prediction model [39]. The multi-layer perception is

the most widely applied neural network architecture today. Neural Network Theory shows that only three layers of neurons are sufficient for learning any (non) linear function combining input data to output data. The input layer consists of one neuron for each complexity metric, while the output layer has one neuron for each quality metric to be predicted.

Because neural network based approaches are predominantly result-driven, not dealing with design intuition or heuristic rules for modeling the development process and its products, and because their trained information is a black-box (that is to say, not accessible from outside). They are not suitable for providing the reasons for a particular result. Therefore, neural networks can be applied when only input vectors (software metric data) and results (quality or productivity data) are of concern, while no intuitive connections are needed between the two sets (e.g. pattern recognition approaches in complicated decision situations).

Most of the prediction models developed using neural networks use back-propagation feed-forward training networks (see Figure 2.5). The network is trained with a series of input and correct output from the training data so as to minimize the prediction error. Once the training is complete, and the appropriate weights for the network arcs have been determined, new input can be presented to the network to predict the corresponding estimate of the response variable.

Most of the models which developed using neural networks operate as "black boxes" and do not provide any information or reasoning about how the outputs are derived.
It is hard to know whether the models satisfactorily predict software quality in different contexts or not.

Therefore we can see that neural networks cannot currently provide any insight into why they arrived at a certain decision rather they only provide the result-driven

connection weights. It is interesting to note that feedforward neural nets can be approximated to any degree of accuracy by fuzzy expert systems [38], hence offering a new approach for classification based on neural fuzzy hybrids that can be trained and pre-populated with expert rules.

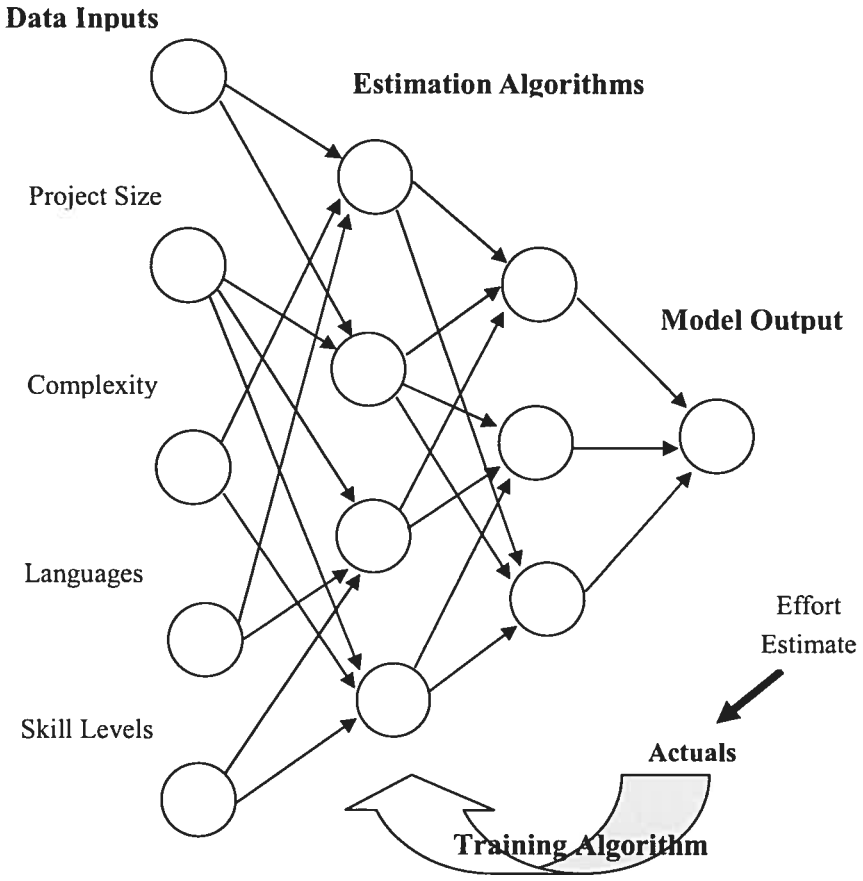


Figure 2.5 A Neural Network Estimation Model

2.5.4 Decision Tree Models

Another kind of prediction model is the decision tree model, also called a rule-based model. A decision tree model is a kind of inductive model that explains the relationship between predictive and predicted variables [57].

A decision tree algorithm is attractive because of its explicit representation of

classification as a series of binary splits (see Figure 2.6). A decision tree algorithm constructs a tree, and the tree can also be translated into an equivalent set of rules. This makes the induced knowledge structure easy to understand and validate.

An empirical decision tree represents a segmentation of the data that is created by applying a series of simple rules. Each rule assigns an observation to a segment based on the value of one input. One rule is applied after another, resulting in a hierarchy of segments within segments. The hierarchy is called a tree, and each segment is called a node. The original segment contains the entire data set and is called the root node of the tree. A node with all its successors forms a branch of the node that created it; the final nodes are called leaves. For each leaf, a decision is made and applied to all observations in the leaf. The type of decision depends on the context. In predictive modeling, the decision is simply the predicted value.

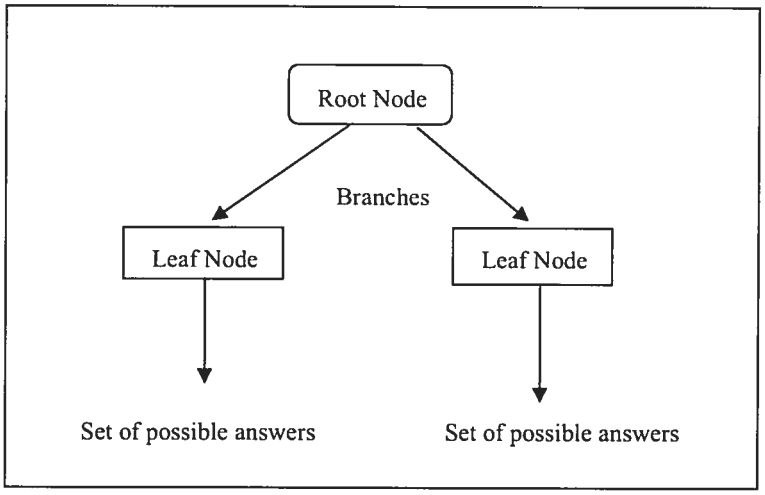


Figure 2.6 A Decision Tree Diagram

In the decision tree:

- Each nonleaf node is connected to a test that splits its set of possible answers into subsets corresponding to different test results.
- Each branch carries a particular test result's subset to another node.
- Each node is connected to a set of possible answers.

A decision tree is a complete binary tree where each inner node represents a yes-or-no question, each edge is labeled by one of the answers, and terminal nodes contain one of the classification labels. The decision making process starts at the root of the tree. Given an input vector x, the questions in the internal nodes are answered, and the corresponding edges are followed. The label of x is determined when a leaf is reached.

More specifically, decision trees classify **instances** by sorting them down the tree from the **root node** to some **leaf nodes**, which provides the classification of the instance. Each node in the tree specifies a **test** of some **attribute** of the instance, and each **branch** descending from that node corresponds to one of the possible **values** for this attribute.

An instance is classified by starting at the root node of the decision tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute. This process is then repeated at the node on this branch and so on until leaf node is reached.

A decision tree is induced from a table of individual cases, each of which describes identified attributes. At each node, the algorithm builds the tree by assessing the conditional probabilities linking attributes and outcomes, and divides the subset of cases under consideration into two further subsets so as to minimize entropy according to the criterion it chooses. The criterion for evaluating a splitting rule may be based on either a statistical significance test or on the reduction in variance or entropy. All criteria allow the creation of a sequence of sub-trees.

Normally, the decision tree is constructed by Quinlan's ID3 algorithm. C4.5 is a software extension of the basic ID3 algorithm designed by Quinlan. This algorithm belongs to the 'divide and conquer' family of algorithms where a decision tree generally represents the induced knowledge. C4.5 works with a set of examples that has the same structure and consists of a number of attribute/value pairs. One of these

attributes represents the class of the example. Most of the time the class attributes are binary and take only the value {true, false}, or {success, failure}. The key step of the algorithm is selecting the "best" attribute so as to obtain compact trees with high predictive accuracy.

An advantage of decision tree models over other models is that this kind of model may represent interpretable English rules or logic statements. For example, "*If monthly mortgage-to-income ratio is less than 25% and months posted late is less than 1 and salary is greater than $35,000, then issue a silver card.*"

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute-values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions. Our algorithm is designed specifically to combine the classic-rule based prediction models for stability into one final classifier. A classic-rule based prediction models is a set of decision tree classifiers (Figure 2.7).

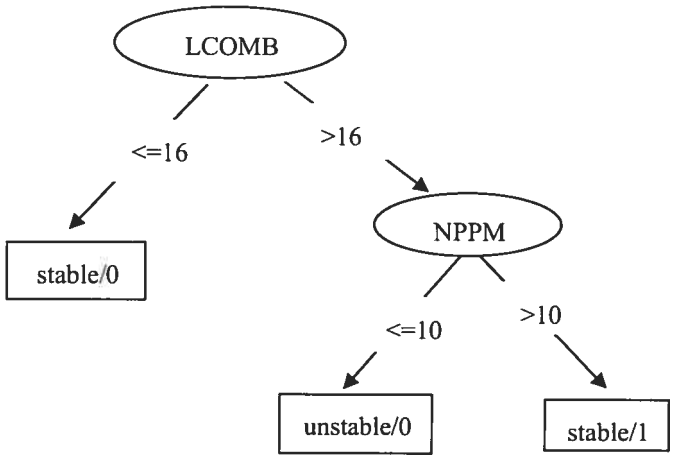


Figure 2.7 A Decision Tree for Stability Prediction

The following example provides a sample rule that is derived from the above decision tree.

LCOMB > 16
NPPM <= 10
→ *class* 0 [63.0%]

Figure 2.8 A Rule Set Translated from Figure 2.7

2.6 Summary of this Chapter

In this chapter we described the basic concepts of software quality. We also introduced the main approaches of building software quality prediction models and some of the existing models. In our research, we will propose a new method –a combination algorithm by using a genetic algorithm – to build new models. We use existing decision tree models (rule based models) as our input and we believe the obtained new models have better prediction ability. In the next chapter we will describe the genetic algorithm in more detail.

Chapter 3 Genetic Algorithm Principles

A genetic algorithm (GA) is an optimization technique that was introduced in the late 60's by John Holland [36]. GAs were inspired by Darwin's theory of evolution. They can be used for applications such as training neural networks, selecting optimal regression models and discriminant (pattern recognition) optimization [22].

A GA imitates the process of creating a new population of individuals. The components of a GA are chromosomes each of which display a certain fitness. The fitness is used to measure how well the individual performs in its environment. The key idea of the Darwinian theory of evolution is that new chromosomes are created and the fittest remain until the end and propagate their genetic material during evolution. The new chromosomes are created through three major operators: selection, crossover/recombination and mutation [22].

In this chapter, we first give a brief introduction to the GA. Then we describe GA concepts: operators and parameters. Finally we present the GA application.

3.1 Introduction of Genetic Algorithm Principles

The scope of GAs is very broad. GAs are a part of evolutionary computing, which is a rapidly growing technique of artificial intelligence [51]. Generally, the process of a GA can be described as follows:

A GA starts with a set of solutions (represented by chromosomes) called the original population. Solutions from the original population are taken and used to form a new population. It is hoped that the new population will be better than the old one. Solutions selected to form new solutions (offspring) are chosen according to their fitness. The more suitable they are the more chances they have to reproduce. This process of reproduction is repeated until certain conditions, for example, the number of generations or the best solution, are satisfied.

The following represents the outline process of a typical GA.

1. **[Start]** Generate a random population of n chromosomes (suitable solutions for the problem).
2. **[Fitness]** Evaluate the fitness $f(x)$ of each chromosome x in the population.
3. **[New population]** Create a new population by repeating the following steps until the new population is complete.
    1. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better the fitness, the better the chance of being selected).
    2. **[Crossover]** Using crossover probability, cross over the parents to form new offspring (children). If no crossover is performed, offspring is an exact copy of the parents.
    3. **[Mutation]** Using mutation probability, mutate new offspring at each locus (position in the chromosome).
    4. **[Accepting]** Place new offspring in a new population.
4. **[Replace]** Use newly generated population for a further run of the algorithm.
5. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in the current population.
6. **[Loop]** Go to step **2.**

This outline of the GA process is only a general one. There are many things that can be implemented in different ways and in various domains. In order to better understand a GA, the following sections provide a more detail description of the GA procedure.

3.2 Terms of Genetic Algorithm

Before going into the details of a GA, some terms associated with it need to be defined to help understand how it works.

3.2.1 Chromosome, Gene and Genome

From the view of biology, all living organisms consist of cells. Each cell contains the same set of **chromosomes**. Chromosomes are strings of DNA and serve as a model for the whole organism. A chromosome consists of **genes** that are blocks of DNA. Each gene encodes a particular protein and a **trait** such as the color of eyes. Possible settings for a trait (e.g. blue, brown) are called **alleles**. Each gene has its own position in the chromosome. The position of a gene is called its **locus**. The genome is the complete set of genetic material (all chromosomes).

GA borrowed several terms from biology. For example, the term **chromosome** refers to one individual element in the search space. A chromosome is formed from genes. Simply speaking, genes are the individual instructions that tell the organism how to develop and keep the body healthy, while chromosomes are the structures that hold the genes. In every cell of an organism there are thousands of genes that are located on each chromosome. Chromosomes occur as pairs. Figure 3.1 shows a pair of chromosomes and a chromosome structure.

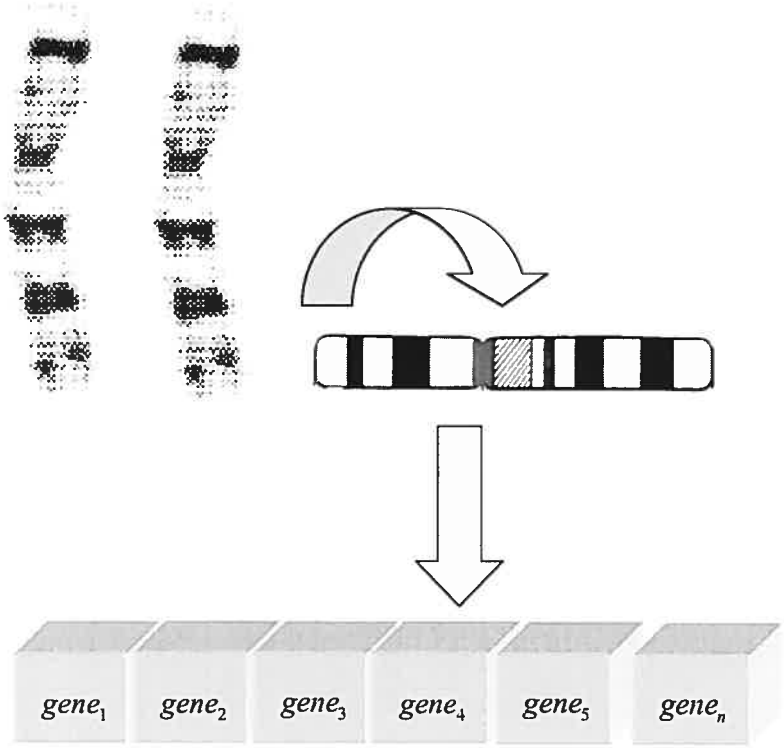


Figure 3.1 Chromosome Pair Nature Shape and Representation in Our Study

In biology, each gene is responsible for a certain trait in an individual. The locus of the gene determines what trait the gene will influence, while the allele of the gene determines how the trait will be influenced. For example, a biological gene occupies the locus " hair color" and the allele "red" then the result will be red hair. In nature this phenomenon is very complex; therefore, the focus will be on how this concept is used in our work and will not be gone into in detail. More information can be found in Wolfgang and Banzhaf's book about Genetic Programming [58].

When they were first introduced, Gas dealt with binary representation and chromosomes with fixed-lengths. Figure 3.2 shows that a chromosome was a binary string.

Figure 3.2 The Binary Representation of a Chromosome

Later on, variations were brought in and chromosomes took different forms such as multiple figures instead of only the binary values [28]. Although a substantial amount of GA research has been done with variable length chromosomes, the majority of GA work is focused on fixed-length chromosomes [28].

3.2.2 Genotype and Phenotype

The **genotype** and the **phenotype** are terms also borrowed from biology. A particular set of genes in a genome is called a genotype. The genotype is the basis for the organism's phenotype, which is their physical and mental characteristic, such as eye color and intelligence.

The concept of a genotype and a phenotype are essential to the understanding of a genetic algorithm. The genotype is the encoding of the information in genetic code, and it is decoded (or interpreted) by several enzymes to construct an individual organism. This individual is the phenotype; that is, it is the actual manifestation of the information contained in the DNA in the genes. Figure 3.3 shows the Genotype and Phenotype.

Genotype (code) Phenotype (individual organism)

Figure 3.3 Genotype and Phenotype in Nature

In this study, the GA only works on the genotype – the encoding of the genetic code. The algorithm itself has no notion of the phenotype. Later on, to test for how well they perform in their environment, the "fitness" of each individual is measured on the phenotype as the individual.

3.2.3 Generation and Population

The idea of generation is similar to that found in nature. The population is a set of individuals (chromosomes). The terms chromosome and individual are interchangeable in referring to one individual element in the GA. All individuals in the original population make up the first generation. GA operations like selection, crossover and mutation (see Section 3.3) are performed on this generation. Pairs of chromosomes are selected to propagate new individuals. All the newly created individuals together make up the second generation. Then through the next operation, comes the third generation, the fourth one and so on, and the population can grow in to a new generation. Therefore, a generation can be thought of as the whole set of individuals whose parents are from the same generation level above them, while the population of a certain generation refers to the total of individuals in that generation.

3.2.4 Fitness

Fitness is a value we assign to a chromosome to measure how well it performs in an environment. The fitness score is a possibility-transformed rating used by the GA to determine the fitness of individuals for mating. The GA uses the fitness scores to

determine selection. The value of fitness usually is between 0 to 1 with 1 being strongest and 0 being weakest. Therefore, better chromosomes have a stronger fitness value. In most of the GA studies, the chance of a chromosome being selected is proportional to its fitness value.

3.2.5 Search Space

When people are solving a problem, they are usually looking for solutions that are the best among all the possibilities. The realm of all feasible solutions is called the **search space** or also known as the state space by some researchers. Each point in the search space represents one feasible solution. Each feasible solution can be evaluated by its fitness value for the problem. Therefore, finding a solution is concerned with locating the extreme fitness (maximum or minimum) points in the search space. However, when solving a problem, people are usually only aware of a few of the points from the whole search space, which means there are many unknown points while other points are generated as the process of finding a solution evolves.

The problem is that the search for a solution can be very complicated. One does not know where to look for the solution or even where to start. There are many methods on how to find suitable solutions, which may not necessarily be the best solutions.

3.3 The Genetic Algorithm Operators

There are three major genetic operators in a GA. By applying these operators to the current generation, a new generation can be created. By running a GA a sequence of evolutions from one population of chromosomes to another is generated. The three major genetic operators, which will be explained in the following sections, are selection, crossover, and mutation.

3.3.1 Selection

Selection is the operator used to select the mating partners. As we have already seen from the concept of a GA, chromosomes are selected from the population to be parents

to create new chromosomes. How to select these chromosomes can be a problem. According to Darwin's theory of evolution, the best chromosomes should survive and create new offspring. This can be done in many ways, but the main idea is always to select the better parents in hope that the better parents will produce better offspring. Many methods have been generated to select the best chromosomes, such as roulette wheel selection, Boltzman selection, tournament selection, rank selection, and steady state selection [17].

- **Roulette Wheel Selection**

In this selection process, parents are selected according to their fitness. The better the chromosomes are, the more chances they have to be selected. Imagine a roulette wheel where all chromosomes in the population are placed. The size of the space for each chromosome is proportional to its fitness values (See Figure 3.4)

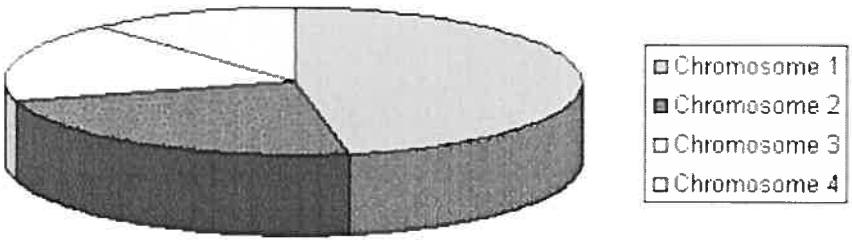


Figure 3.4 Roulette Wheel

Then a marble is thrown on the wheel and whichever chromosome's position it stops on, the chromosome will be selected. Obviously the chromosomes with stronger fitness values are more likely to be selected.

This can be simulated by the following algorithm.

1. **[Sum]** *Calculate the sum of all chromosome fitness values in a population - sum **S**.*

2. **[Select]** *Generate a random number from interval **(0,S)** - **r**.*

3. **[Loop]** *Go through the population and sum fitness values from **0** - sum **S**. When the sum **s** is greater then **r**, stop and return the chromosome from*

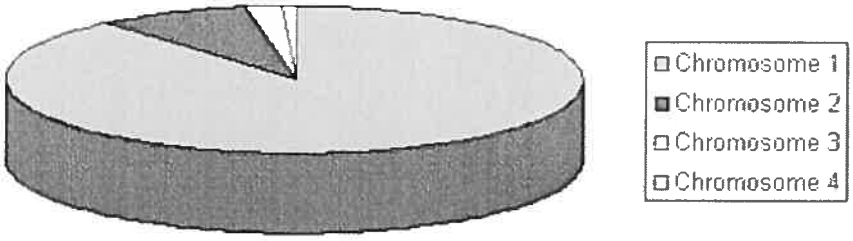
where you are.

Of course, step 1 is performed only once for each population.

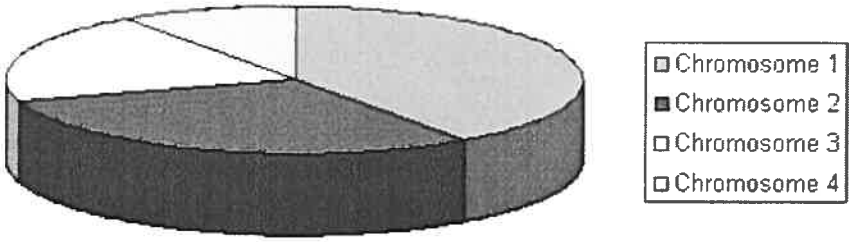
- **Rank Selection**

The roulette wheel selection process will have problems when the fitness values differ a lot. For example, if a chromosome with the best fitness value occupies 90% of the whole roulette wheel then the other chromosomes will have very few chances to be selected. Therefore, rank selection addresses this problem by first ranking the population with a sequence of increasing fitness values. Then every chromosome has a rank number from this ranking. The one with the worst fitness value will have rank number 1; second worst rank number 2 etc. The best one will have a rank number N (number of chromosomes in the population). Then these chromosomes go on the wheel according to their rank numbers.

The following diagrams (Figure 3.4) demonstrate how the situation changes between fitness proportion and rank number.



Situation before ranking (graph of Roulette Wheel)



Situation after ranking (graph of order numbers)

Figure 3.5 Rank Selection

After this change the chromosomes with lower fitness values have a greater chance of being selected. But this method can lead to slower convergence, because the best chromosomes are not as distinguishable from the others.

- **Steady-State Selection**

This method is not specific to parent selection. The main idea of this selection process is that the best part of the chromosomes should always survive to the next generation.

The GA then works in the following way. In every generation a few chromosomes with strong fitness values are always selected to create new offspring. Then some of the chromosomes with the lowest fitness values are removed and the new offspring takes their place. The rest of the population survives to the new generation.

3.3.2 Crossover

Crossover, or in some cases it is know as recombination, is the most important genetic operator. Analogous to the biological process, this operator captures the process when two chromosomes bump into each other, exchange some of their genes, give birth to two new offspring and then drift apart. Each of the new offspring inherits traits (pieces of information) from both its parents. In nature, we can see this in a human baby when it takes the skin color of the father and the eye color of the mother. Fitter individuals in a particular generation have a higher probability of undergoing crossover and producing progeny. It is this operator that causes evolution since the idea behind it is to combine in one individual all of the "good" traits, in order for these traits to disperse in the whole population, hence create "better" individuals. In a GA, the same process is simulated; however, the exchange of genes can happen in many different ways. The method used for this study will be described in Chapter 5. Like in nature, crossover does not always occur for all selected couples. The probability for crossover to occur within a selected couple is usually between 80% and 90%. In many cases, a probability between 50% and 60% is found to be the best [22]. If crossover does not occur GA for a couple, the offspring are exact copies of their parents.

After deciding which encoding to use, the operation of crossover can be carried out. In classical GAs, the representation of a chromosome is a bit-string. The **cutting point**, which decides which genes are to be exchanged, is randomly chosen and the chromosome length is fixed. The simplest method of doing this is to copy everything before this point from one parent and everything after this point from the other parent. For example: In the following figure, we consider two chromosomes that have a length of 15 each. The first chromosome has genes with values of all 1's and the second one has genes with values of all 0's. Crossover is done after the fourth gene in each chromosome.

Figure 3.6 shows how this method works. (The crossover **cutting point** is marked with more space):

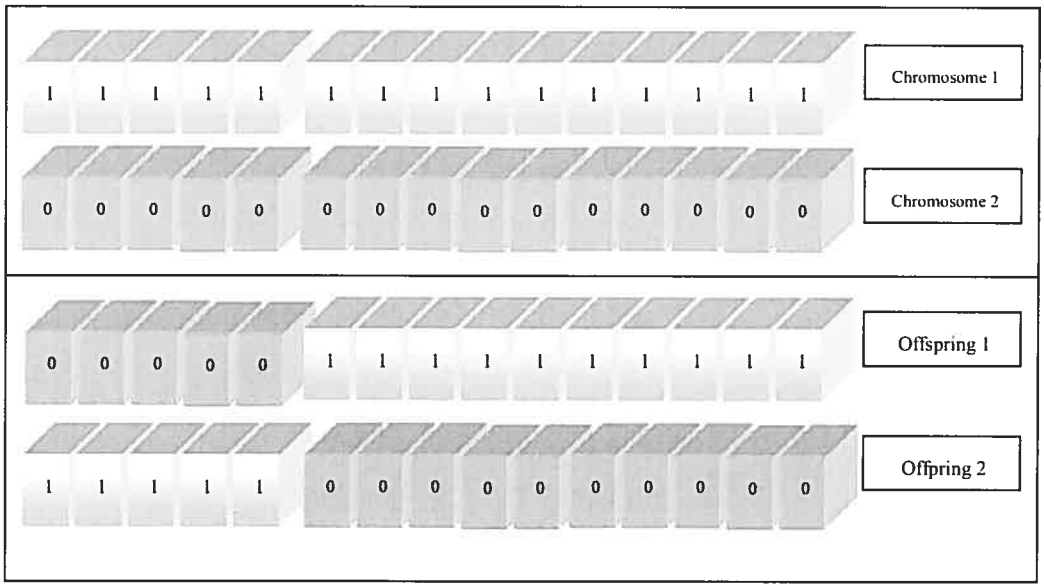


Figure 3.6 Crossover (cutting point 5, fixed length)

This is the simplest way to perform crossover. However, crossover can be done on more than one cutting point and the length of the chromosome can vary. Crossover can be complicated and very dependent on the encoding of the chromosome.

3.3.3 Mutation

Mutation is another major genetic operator. For this operator, a randomly chosen gene within a chromosome, for certain reasons may be changed to a random value from the domain of values for the genes. For example, in the bit-string representation, only two values (0,1) are possible. If the value of a gene is changed during the crossover, we say mutation occurs. In nature, duplicating DNA can sometimes result in errors while the genetic information is copied from the parents to the next generation. DNA is also prone to damage in day-to-day existence [27]. In GA, the idea behind simulating mutation is to stop the algorithm from being stuck at local optima.

A proper probability for mutation in GA needs to be carefully set. If the probability of mutation is very high, the algorithm will turn into a random search, which is inefficient to find good chromosomes. Typically, the probability for a gene to be mutated ranges between 0.1% and 10% [22].

Mutation might take place after crossover is performed. This is to prevent all solutions in the population from falling into a local optimum of solved problems by bringing in some new genes. Mutation randomly changes the new offspring. The following (Figure 3.6) shows an example of mutation on a chromosome. For binary encoding a few randomly chosen bits can be switched from 1 to 0 or from 0 to 1. Mutation can occur as follows (mutation occurs in the fourth and the last gene):

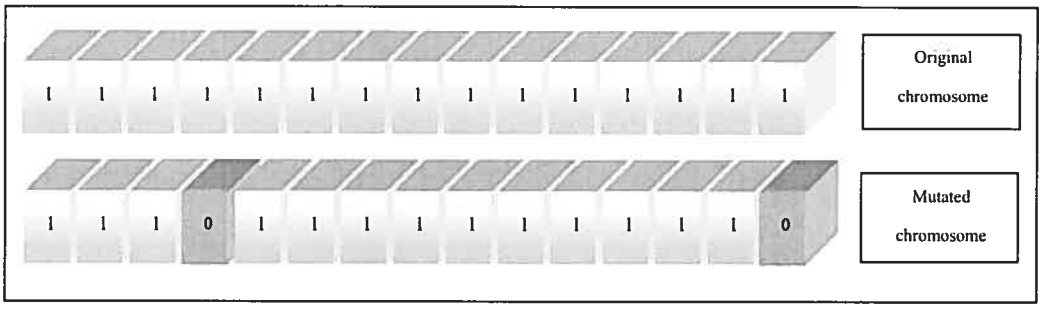


Figure 3.7 Mutation

Mutation depends on the encoding as well as the crossover. For example, when we are encoding permutations, mutation can be exchanging two genes.

3.3.4 Elitism

Creating a new population only through crossover and mutation can result in the loss of the best chromosomes from the present population. Therefore, a method called **Elitism** is often used. This means, that at least one of the best solutions is always copied, without changes, to the new population, in order to ensure that the best solution survives to end of the run. This is done by copying the best chromosome (or a few of the best chromosomes) to the new population. The rest undergoes to the normal crossover and mutation operations. Elitism can very rapidly increase the performance of the GA, because it prevents the loss of the best-found solution.

3.4 Parameters

There are several important parameters in GAs. The three basic parameters of a GA are **population size**, **crossover probability** and **mutation probability**. The following provides a brief introduction to them.

3.4.1 Population Size

Population size is an important parameter in GA. We especially care about the population number of one generation. It determines the maximum number of chromosomes in a generation used to create new offspring. If there are too few chromosomes, GA has few possibilities to perform crossover and only a small part of the search space is explored. On the other hand, if there are too many chromosomes, the GA process is slowed down. Research shows that after a certain limit (which depends mainly on encoding and the problem) it is not useful to increase the population size, because it does not make solving the problem more efficient [22].

3.4.2 Crossover Probability

This parameter decides how often crossover will be performed. If there is no crossover, the offspring will be an exact copy of the parents. If there is crossover, the offspring is recombined from parts of each of the parents' chromosomes. That is to say, if crossover

probability is **100%**, then all offspring is made by crossover. If it is **0%**, the whole new generation is made from exact copies of the chromosomes from the old population (but this does not mean that the new generation is the same). Crossover is performed in the hope that the new chromosomes will carry the beneficial parts of the old chromosomes and perhaps even the new chromosomes will be better. However it is a good idea to maintain some part of the old population in the next generation as proposed by Elitism.

3.4.3 Mutation Probability

Mutation probability refers to how often the parts of a chromosome will be mutated. If there is no mutation, the offspring is determined after crossover without any change. If mutation is performed, parts of the chromosome are changed in the next generation. If the mutation probability is **100%**, the whole chromosome could be changed; if it is **0%**, nothing is changed. Mutation is performed in order to prevent the GA from falling into a local extreme, but it should not occur very often, because then the GA will in fact turn in to a random recombination.

3.5 Three Stages of a Genetic Algorithm Application

When the GA is applied to solve a problem, normally, there are three distinct stages [42]:

1) Problem Representation.

This deals with how the potential individual solutions of the problem domain can be encoded into a representation that supports the necessary variation and GA operations. These representations are often as simple as bit strings (Figure 3.2). A good representation can make the problem easy to understand and deal with.

2) **Genetic Algorithm Operation.**

In the second stage, analogous to the sexual activity of biological life forms, a GA applies mating and mutation algorithms so as to produce a new generation of

individuals. The new generation recombines features of their parents. Three GA operators, selection, crossover and mutation, are used to produce the new generations.

3) Fitness Function.

A fitness function judges which individuals are the "best" life forms, that is, most appropriate for the eventual solution of the problem. These individuals have more chances of survival (reproduction) and shaping the next generation of potential solutions. In our algorithm, we purposely always copy the individual with the highest fitness in this generation to the next generation; therefore the best individual will not be lost. Eventually, all individuals of a generation will be referred back to the original problem domain as a solution for the problem, and the fitness value is assigned to each of them.

3.6 Summary of this Chapter

A GA was inspired by Darwin's theory of evolution. The process of evolution starts with a set of chromosomes. There are three major genetic operators: selection, crossover and mutation. The fitness function evaluates the suitability of each chromosome. The more suitable a chromosome is the more chance it has to survive and reproduce. Elitism can be used to avoid losing the best suitable chromosome during evolution. Three parameters of a GA, which are population size, crossover probability and mutation probability, affect the efficiency of the evolution.

Chapter 4 Combination Algorithm

In general, the software quality prediction models are obtained from historical measurement data or domain specific heuristics of experts. Unfortunately, not all the software organizations keep their historical data, which makes it difficult to build efficient prediction models.

As mentioned before, many prediction models have been proposed in the last few decades. These models can only accurately predict some aspects of software quality, or they can only satisfactorily work for the specific circumstances from which they were built. Meanwhile, the development of GAs offers a new approach to build prediction models. In this chapter, we propose to build software stability prediction models by combining existing prediction models from various contexts using a GA. Our goal is to verify if this approach can produce a generally applicable model for software quality prediction.

First, we will describe our research methodology. Then we will introduce the data environment and the model encoding of our algorithm. After that we will present and illustrate how the GA works in our domain.

4.1 Research Methodology

This research, generally speaking, uses existing rule-based prediction models (refer to Section 2.6.4 in Chapter 2) as input for recombination by applying a GA. The basic

idea of our research is to start from a set of initial solutions (set of models) to derive new and possibly better solutions.

The following represents a brief introduction to our research:

The derivation starts with an initial solution set P_0 (called the initial population). Then a sequence of populations $P_1 \ldots P_t$ is generated. Each generation is obtained by "recombining and mutating" the previous one while keeping its elitism. Each model of the solution sets is called a chromosome. The fitness of each chromosome is measured by an objective fitness function. Each chromosome (prediction model) consists of a set of genes (prediction rules). At each generation, the algorithm selects certain pairs of chromosomes using a selection method that gives priority to the fittest chromosomes. To each selected pair, the algorithm applies two operators, crossover and mutation, with probability p_c and p_m respectively. Here p_c means the crossover probability and p_m the mutation probability. Both of them are input parameters of the algorithm. The crossover operator mixes the genes of the selected chromosome pair, while the mutation operator randomly changes certain genes. Each selected pair of chromosomes produces a new pair of chromosomes that constitute the next generation. The fittest chromosomes of each generation are automatically added to the next generation to keep the elitism. The algorithm is completed when a convergence criterion is satisfied or when a fixed number of generations are reached. At the end, we analyze the best model generated from the evolution and compare it with the initial ones. If it is better, then we can conclude this approach is applicable.

4.2 Data Environment

In this study, the classic rule-based prediction models are presented as the chromosomes. The rules in each model are the genes. We use a metrics database file as the training and testing environment in which the metrics value and the real classifier value are given.

The term "model", "prediction model" or "chromosome" mentioned below indicate the classic rule-based prediction model only. An example of the prediction model structure is shown below (Figure 4.1). Class 1 indicates the software is stable while class 0 indicates instability.

```
Model Model 1:
  Rule 3:
    coh <= 0.033735
    COM <= 0.15789
    OCMAIC > 2
    OCMAIC <= 4
    CUBF > 4
    CUBF <= 7
    → class 0  [73.3%]

  Rule 1:
    OCMAIC <= 2
    → class 1  [97.1%]

  Rule 6:
    coh > 0.033735
    OCMAIC <= 10
    → class 1  [93.5%]

  Rule 13:
    coh > 0.033735
    COMI <= 0.16667
    → class 1  [93.0%]

  Default class: 1
```

Figure 4.1 A Classic Rule-based Prediction Model for Stability

This model contains five rules. In the first line, "Model Model 1" indicates the beginning of a model, and the model's name is "Model 1".

The five rules of this model contain four basic rules and one default rule. Each basic rule has a rule name, a set of conditions, a conclusion and a possibility. The last one is the default rule. Figure 4.2 gives an example of "Rule 13" of "Model 1".

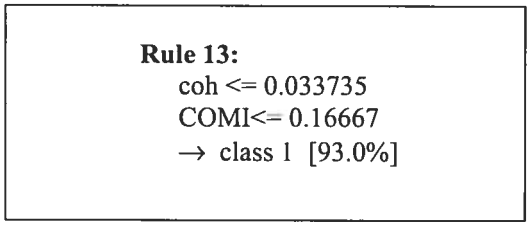


Figure 4.2 An Example of a Basic Rule (Gene) in Model 1

The first line of the rule is the ID of the rule, or the rule name. Figure 4.2's rule name is "Rule 13". Following the name are the two condition sets:

coh <= 0.033735 and

COMI<= 0.16667

If the two conditions are satisfied then the conclusion is realized. The conclusion is followed by the arrow sign (→). For this example it is "class 1" which indicates stability. The 93.0% indicate the truth value. That is, if the condition is true there is a 93.0% probability that the conclusion is true. Therefore, "Rule 13" of the model "Model 1" can be explained as:

If the value of coh is greater than 0.033735 and the value of COMI is less or equal to 0.16667, then this software has a 93% probability of being stable (class 1)

The other three basic rules in model "Model 1" are "Rule 3", "Rule 1" and "Rule 6" and can be understood similar to "Rule 13."

In a model, each basic rule makes a prediction according to the threshold value of some specific metric. If none of the basic rules is applicable, then the default rule is applied. The default rule is the last one in a model. It simply assigns a value to the predicted variable. For example in "Model 1" in Figure 4.1, the last line is "**Default class: 1",** which is the default rule. It can be explained as:

If none of rule 1, rule 3, rule 6 or rule 7 is satisfied, then it predicts the software to be stable (class 1).

The default rule is a special one. It only contains a conclusion and its condition is that no other rules in this model are applicable. It has no possibility value or rule name.

Our study focuses on the classic rule-based prediction models. In general, the classic rule based prediction model's structure is as follows.

<Rule_Set> ::= Rule_Set <RuleSetName><RuleList>$

<RuleSetName> ::= RS<GenerationNumber><SeriesNumber>

<GenerationNumber> ::= [00 to 99]

<SeriesNumber> ::= [00 to 99]

<RuleList>::= <Rule>|<RuleList>;<Rule>

<Rule> ::= <RuleName><CondistionList> -> <Conclusion>

<Rule> ::= <Default Rule>

<ConditionList> ::= <Condition>| <Condition><ConditionList>

<Condition> ::= <MetricsName><ComparationOperator><Value>

<MetricsName>::=[coh | LCOMB | COM | COMI | OCMAIC | CUBF | CUB | OMAEC | NOC | NOP | NON | NOCONT| DIT | MDS | CHM | NOM | WMC | MCC | DEPCC]

<ComparationOpersator>::= [=| <| <=| >| >=]

<Value>::= [int|float]

<Conclusion> ::= class <ClassificationNumber>

<ClassificationNumber> ::= [0|1]

<Default Rule> ::=[Default class: 1|Default class 0]

Figure 4.3 The Rule Based Prediction Model Structure

After understanding the chromosome structure and the data environment in our research, the next step is how to encode the models, which will be introduced in the following section.

4.3 Model Encoding

Our encoding is to define a representation of the classic rule-based models as the chromosome that can be used by a GA. This process includes defining the representation of the model for crossover operation and the representation of a rule for a mutation operation.

4.3.1 Representation of Models

There are different kinds of encoding techniques that have already been used with some success, such as binary encoding, permutation encoding, value encoding and tree encoding. The choice of an encoding technique depends heavily on the problem.

Our problem can't be easily encoded as bit level representations, since the rule set representation in a model is not binary. As mentioned before, in our GA, a chromosome is a model, which consists of a set of rules. Each rule represents a gene. So we use a value encoding method to represent the model. That means each rule was thought of as a value. What follows is a detailed process of shifting from a rule-based prediction model to a chromosome.

First let's review a chromosome's structural representation. In general, a chromosome structure is as follows:

Figure 4.4 A Chromosome Internal Structure in Biology

It can be seen that in the chromosome the genes are arranged in a line with a sequence. The model's representation simulates the chromosome's structure in biology. According to the above chromosome structure, the example model "Model 1" in Figure 4.1 can be represented as follows:

Figure 4.5 The Representation of "Model 1" as a Chromosome

In this chromosome (model), there are five genes (rules). The first four genes (rules) are basic rules with the same structure. The last gene, in the darker color, is the special gene (default rule) with a different structure from the basic rules. Please notice that the sequence of the rules in chromosome is based on their appearance in the models created by C4.5 algorithm. Therefore they may not be sequential.

As previously stated, in this GA a chromosome is a model. Each model is a rule set. Each rule is a gene as illustrated in Figure 4.6.

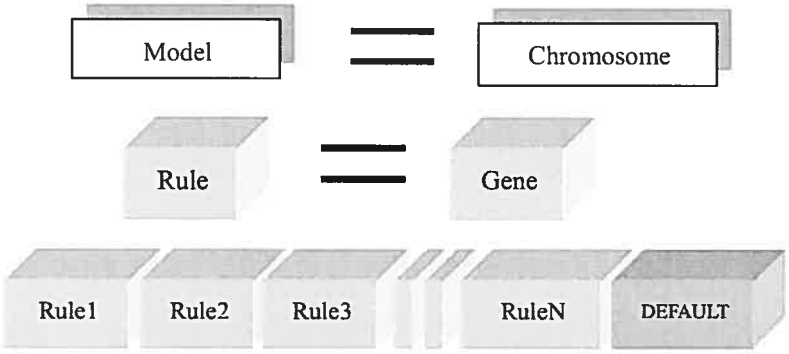


Figure 4.6 Representations of a Chromosome and its Genes by a Model

Now we have the chromosome representation of the models. We can apply the crossover operator to it, but cannot apply the mutation operator because this representation is missing some details of the rule set. To do this, we need to represent the rule set in the chromosome. Because there are two different type of rules in a model, the representation of them will be different too.

4.3.2 Representation of Basic Rule and Default Rule

As illustrated in Figure 4.6, genes (rule) named "Rule 1", "Rule 2", "Rule 3", ...and "Rule N" etc, which are in the lighter color, are basic rules with a basic rule structure (see Figure 4.7).

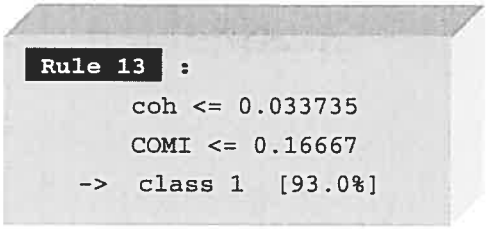


Figure 4.7 Example of the Internal Structure of a Gene (for basic rule)

In general, a basic rule consists of one or more conditions and a conclusion. It can be represented as follows (Figure 4.8)

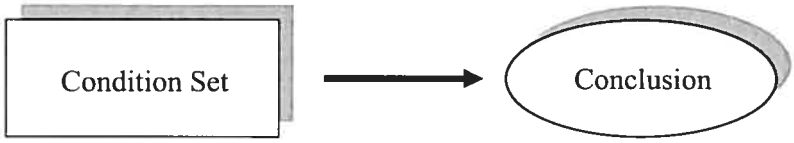


Figure 4.8 A Basic Rule Structure

Furthermore, a condition typically compares the numerical value of the structural metrics to a threshold value. So the condition can be represented as following Figure 4.9. Therefore the mutation operation can be performed by changing the threshold value (add/minus a reasonable value which can be called as step value).

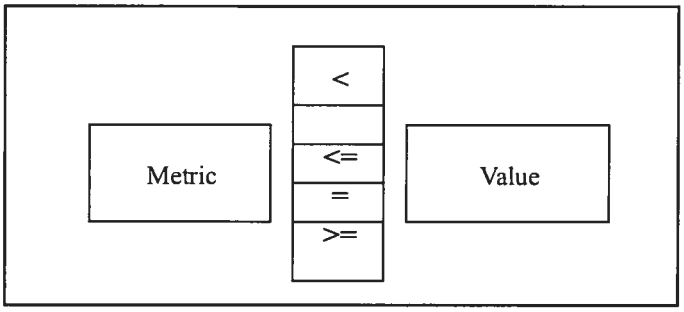


Figure 4.9 Structure of a Condition

Within a gene, if all conditions are true, a value is assigned to the quality characteristic. In our context, the conclusion is either one of the following:

Class 1: indicate software is stable.

Class 0: indicate software is unstable.

The last gene in Figure 4.6, in the darker color, is the special one, named DEFAULT, which we call the "default rule ". In our algorithm, the default rule only takes the following two values.

$DEFAULT_1$ represents the default rule. "Default class: 1".

$DEFAULT_0$ represents the default rule "Default class: 0".

Meanwhile $DEFAULT_{\{1/0\}}$ indicates $DEFAULT_1$ or $DEFAULT_0$.

The default rule has only one of the following conclusion values but without conditions:

Default class: 0 or *Default class: 1*

The representation of the chromosome is very important for the definition of the GA operators. The mutation operation will depend on the range of the threshold value and conclusion value of each gene. The cutting point which selects for crossover will depend on the length of all the chromosomes. Because of the various numbers of rules in each model, the chromosomes in our algorithm will have different lengths, which affect the crossover operator.

4.4 Initial Generation

In order to apply the GA, the initial generation should be obtained first. The initial generation of our algorithm consisted of a set of classic rule-based models. Such models can be collected from the published paper or created by some other algorithms such as C4.5. In Appendix A we give out all the models of the initial generation used in our experiment.

Our algorithm starts with the initial generation applying GA operators to obtain the new generation. Then we treat the new generation as the current generation to create the next generation and then the latest generation is the current one. This process is looped in

that it keeps creating new generations until the terminal condition is matched.

4.5 Combination Algorithm Operators

As introduced in chapter 3, the GA runs by performing selection, crossover and mutation to produce offspring. Crossover and mutation are primarily the most important parts of the GA. These two operators have the main influence on the performance. In the following section we introduce how the GA operators are defined in our study.

4.5.1 Selection

Although there are several selection methods (see Chapter 3, Section 3.3.1) available, for this domain, we choose the roulette wheel method. We use an array RouletteWheel[0:9999] to simulate the roulette wheel depending on each model's fitness (see section 4.6 for the definition of fitness). A percentage value is assigned to each model according to its fitness based on the following formula:

$$P_i = \frac{fitness_i}{\sum_{n}^{1} fitness_j}$$

Where $fitness_i$ indicates $Model_i$'s fitness value and P_i indicates the $Model_i$'s percentage. Figure 4.10 illustrates the roulette wheel used in our algorithm. Each model occupies a certain room in the wheel according to its fitness value.

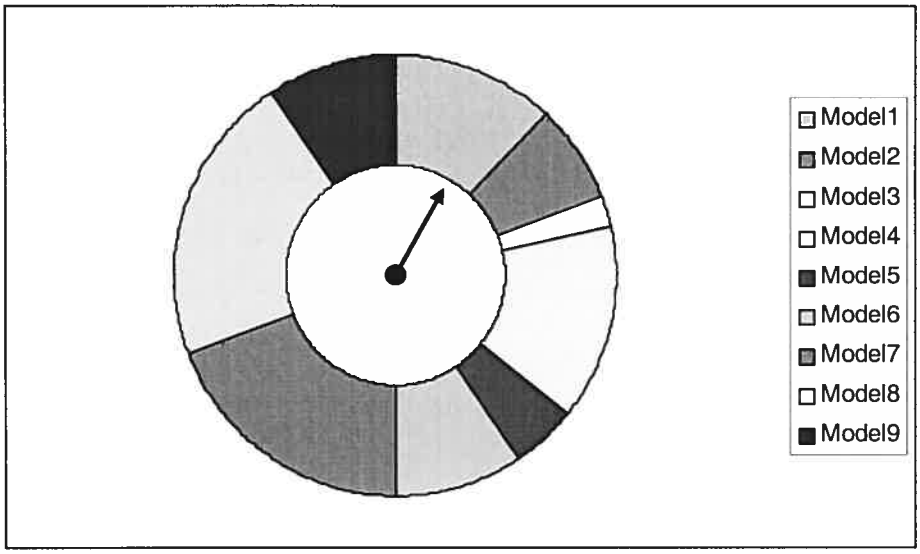


Figure 4.10 Roulette Wheel for Selection

The percentage value of each model determines the portion of the wheel it covers. For example, if $Model_1$'s percentage is *2.3%,* we will assign *2.3%X10000 =230* to be the name $Model_1$. Then RouletteWheel[1] to RouletteWheel[*230*] = "$Model_1$". Then we take the second $Model_2$, assuming its percentage is 3%. Since *3%X1000 =300*, then the array elements from *231* to *230+300=530* are assigned to it and given the value "$Model_2$". This process goes on until all models are assigned. Obviously the higher fitness values of a model has the higher percentages, and the larger portion it occupies on the Roulette Wheel. Therefore when selection is performed, these models have a greater opportunity to be chosen.

After the assignment of percentages, all models' names are spread over the 10000 array elements. we use a random function to get a value between 0 to 9999 to simulate a roulette wheel. The selected number will indicate which model is selected. For example, if the random function selects number *345*, we take out the array value RouletteWheel[*345*]. It is "$Model_2$", then we take $Model_2$.

For each selection process, we ran the random function twice in order to get a pair of chromosomes from the current generation. One of the selected chromosomes is called the father and the other one the mother as the following function shows. The input parameter is the current generation and the outputs are a pair of chromosomes selected to crossover.

$$Selection\ (Generation_n) = (Chromosome_{father}, Chromosome_{mother})$$

The number of selected pairs is half of the generation population size. For example, if the population of this generation size is 50, then we select 25 pairs; if the population size is 51, we select 26 since 25.5 is rounded up to 26. Because each selected parent pair creates two children, then the next generation size will gradually increase.

4.5.2 Crossover

Before crossover can be conducted, its starting position -- the cutting point - must be decided first. During our study, the cutting point is an input parameter. The cutting point can be one or two. The cutting point will be applied to one entire generational loop without changing its value. The input cutting point parameter is set by the following classes:

- The class sets one cutting point: *SetCutPoint(n)*.
- The class sets two cutting points: *SetCutPoint(m, n)*.

The cutting point needs to be set for each GA application. However if a chromosome has few genes, we should ensure the cutting point is not over the length of the chromosome. In this case, the algorithm will change the cutting point to the possible value.

After the cutting point is set, crossover is very simple: each pair of chromosomes switches genes before the cutting point, and keeps the same genes after the cutting point. Then two new chromosomes are created and they are the recombination of their parents. The following describes in detail these two crossover methods.

- **One Cutting Point Crossover**

Using one cutting point to perform the crossover of two chromosomes is the simplest way to produce offspring. Although any position can be the cutting point, it's better to select a cutting point that is within all the chromosomes. Some of our input models have three genes only, that is to say they have the length of three. Therefore we should not take the cutting point greater than three.

Another special issue that should be considered is the last gene. The last gene is the default rule. This kind of gene should not appear twice in one chromosome but each chromosome must have only one default rule. To avoid missing or having multiple default rules during the crossover, the cutting point must be put before the last gene (the default rule). The technique we used to solve this problem is to calculate the length of a model by calculating the number of basic rules. Then we make sure the cutting point is not after the default rule.

Figure 4.10 illustrates the one cutting point crossover process. Suppose we select "Model 4" and "Model 13" as the parents and the cutting point is set as two. First, two genes from Model 4 are copied, then the three genes after the cutting point of Model 13 are added to make a new model, named Model 4_1. The same process is applied to get another new model "Model 13_1".

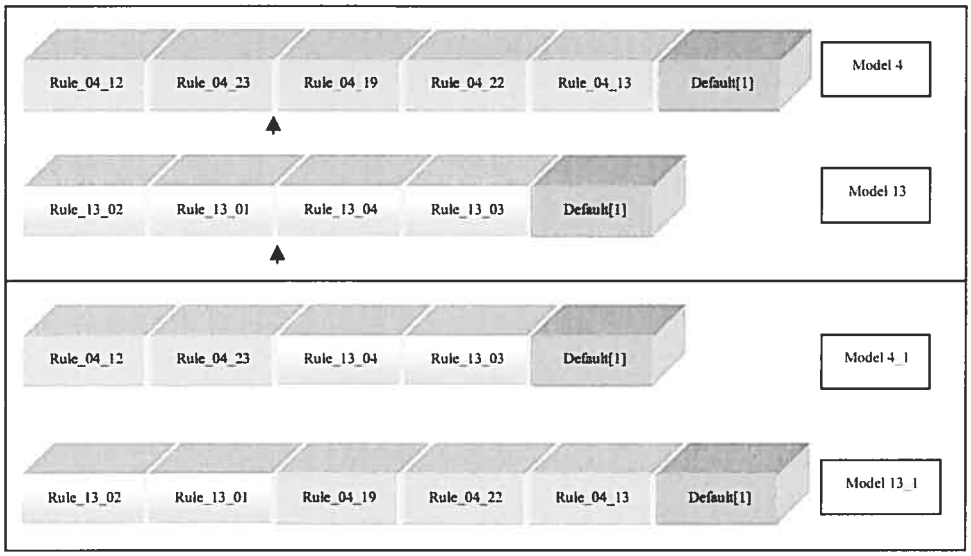


Figure 4.11 Crossover of Model 4 and Model 13 with One Cutting Point

The two real models before doing crossover and after doing crossover are shown in Figure 4.12 and Figure 4.13 on the next two pages.

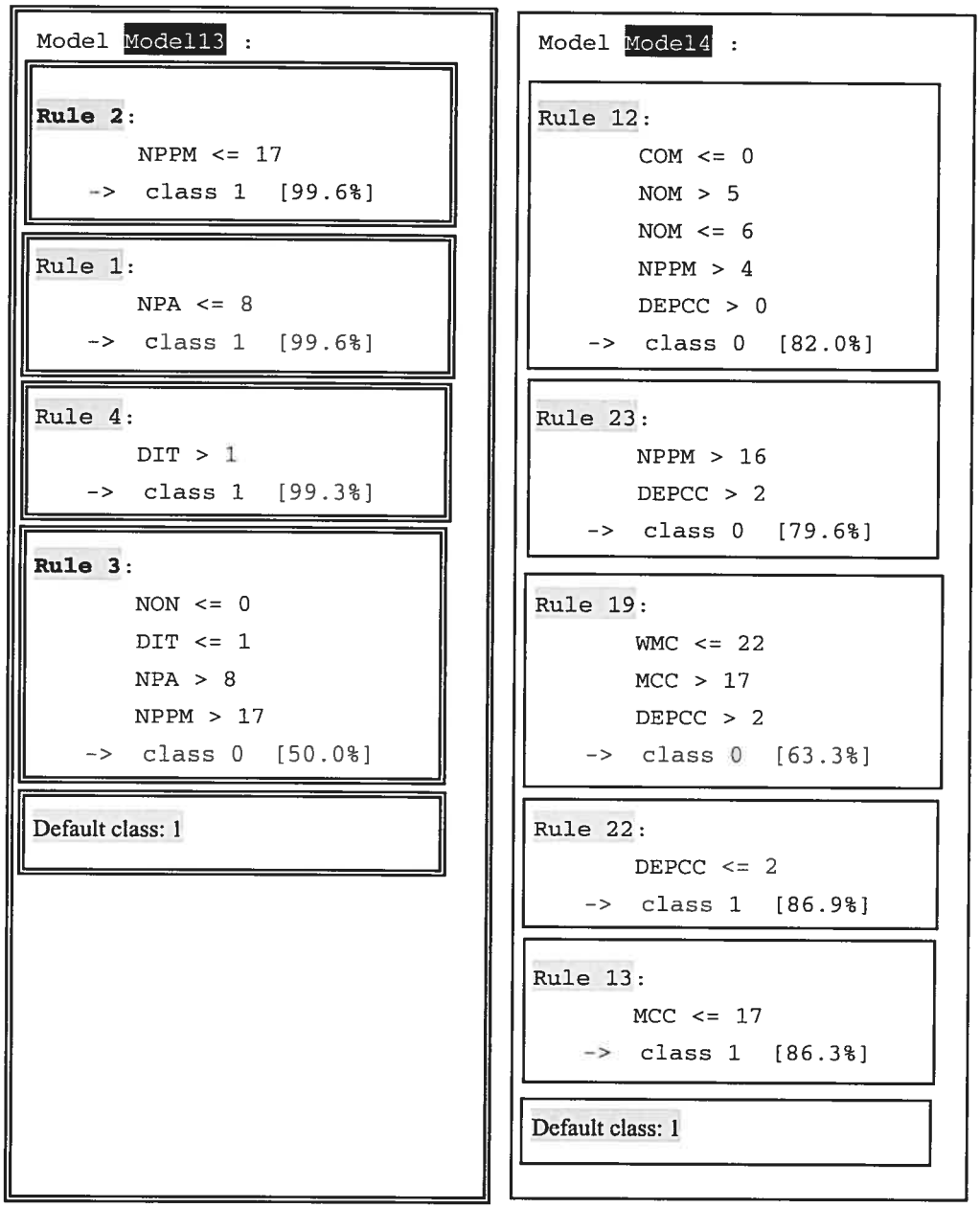


Figure 4.12 Two Original Models ("Model 4" and "Model 13")

In the black box is the model's name (Chromosome name). In the shaded box is the rule's name (gene name) and each box indicates a gene (a rule).

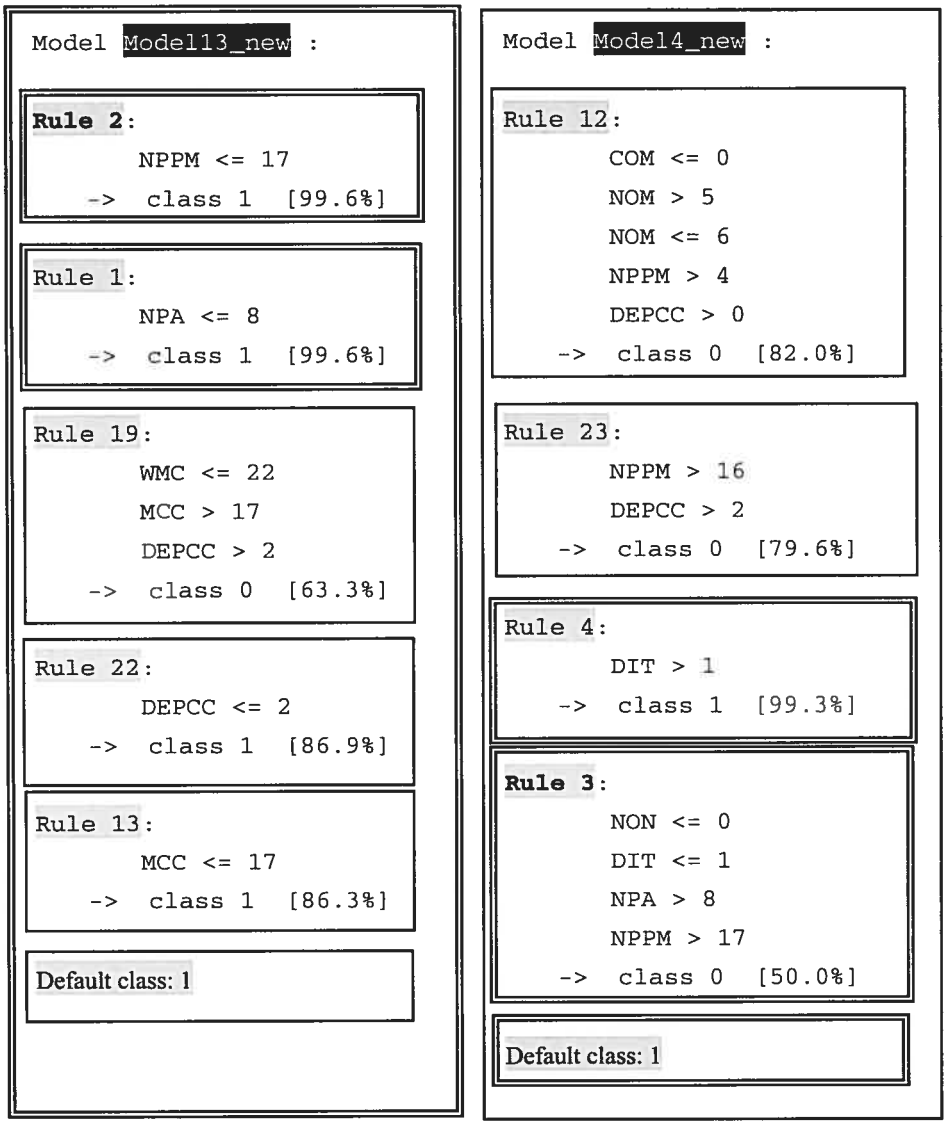


Figure 4.13 Two New Models after Crossover ("Model 4_new", "Model 13_new")

Two Cutting Point Crossover

There can be more than one cutting point for the crossover process. By having more than one cutting point a highly efficient mixing process is created. In our implementation, the GA can perform crossover at two cutting points.

The process of two cutting point crossover is similar to the one cutting point process. Suppose the first cutting point is P_1, and the second cutting point is P_2, the process will switch the genes located in between the two cutting points (the middle part) and keep the head and tail parts. That is to say, all the genes before P_1 or after P_2 in one parent are copied to the new chromosome, and the middle of the new chromosome copies the genes between P_1 and P_2 from the other parent. Figure 4.14 illustrates this process.

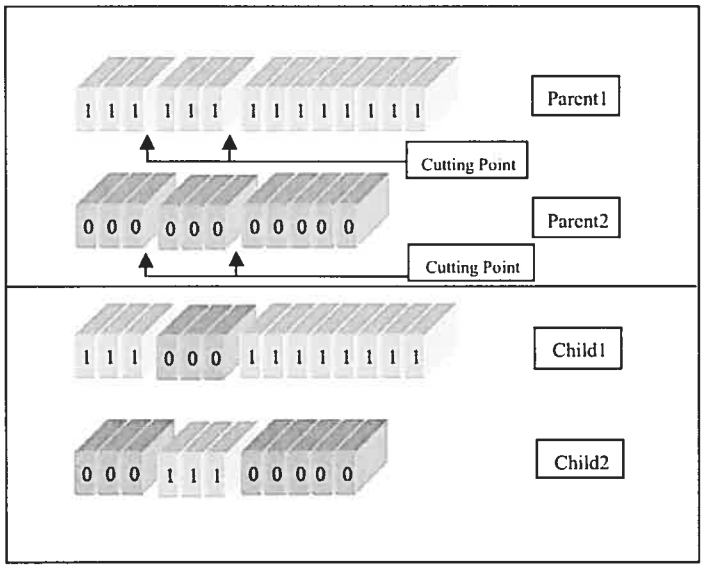


Figure 4.14 Two Cuttings Point Crossover

- **Crossover possibility**

We set the crossover possibility ($Rate_{crossover}$) as another input parameter of operator crossover. Before running the GA, this possibility should also be set. Simulating nature, the possibility for crossover to occur within a selected couple is usually between *80%*

and *90%*. After the couple of chromosomes are selected, our algorithm will decide whether to perform the crossover or not. The implementation of this **Crossover Probability Checking Procedure** is in Figure 4.15. This checking procedure guarantees that the higher the crossover possibility the higher the chance to perform crossover. If $Rate_{crossover} = 1$, then all the selected couples will perform crossover; if $Rate_{crossover} = 0$, crossover will never happen.

1. Produce a random value ($R_{crossover}$) by random function. This value should be between 0 and 1.
2. Make a comparison between $R_{crossover}$ and the Crossover possibility $Rate_{crossover}$.
3. If $R_{crossover} \leq Rate_{crossover}$ *then* {perform crossover operation}.
4. If $R_{crossover} > Rate_{crossover}$ *then* {no crossover happens, the offspring are exact copies of their parents.}

Figure 4.15 Crossover Probability Checking Procedure

4.5.3 Mutation

Mutation is another important operator in a GA. After the crossover operation, a mutation can occur to the genes depending on the *Mutation Probability*. As we have introduced in Chapter 3, this operator randomly chooses genes from a chromosome and gets its value perturbed to a random one from the domain of possible values. The idea of mutation in this GA is to stop the algorithm from getting stuck at a local optima. The Mutation Probability should not be set very high otherwise the algorithm will become a random search. Typically, the *Mutation Probability* is set between the range of *0.1% and 10%*.

The *Mutation Probability* in our implementation will be treated as another input parameter. In our GA we set the *Mutation Probability* as 5% or 10%. That is to say, after crossover is performed, for every five (or 10) out of 100 pairs of chromosomes mutation is possible.

After a crossover is done, the algorithm needs to decide whether the mutation should occur or not to the two newborn chromosomes according to the *Mutation Probability*. The checking procedure is very similar to the crossover probability checking procedure in Figure 4.15 in that the $Rate_{crossover}$ is simply replaced by the *Mutation Probability*.

If mutation is decided upon, first, our algorithm needs to randomly choose one rule from the newborn chromosome, then randomly choose one condition from this rule. The mutation applies only to this condition. Because mutation will change the gene to a random value from the value domain of the genes, we have to define the domain to implement this. Due the fact that our algorithm focus on the classic rule based prediction models only, the mutated genes should be reasonable in such a model after the mutation.

From Section 4.3, we have seen that in our domain there are two kinds of genes. One is a basic rule gene, and the other is a default rule gene. Therefore, there are different operations according to the different rule types or genes. The following is a detailed description for the mutation operator implementation:

- **Mutation on Basic Rule Gene**

This kind of gene consists of conditions and conclusions as showing in Figure 4.9. The mutation operation only performs on a condition set in our algorithm to simplify the implementation. Suppose we mutate the following rule as an example.

Rule 19 :	*Gene Name*
WMC <= 22 MCC > 17 DEPCC > 2	*Condition set*
-> class 0 [63.3%]	*Conclusion*

Figure 4.16 An Example of Rule with Structure Illustrated

For this kind of gene, the mutation is performed on the condition set. Furthermore, the change on the condition is only to increase or decrease the metric threshold value. The reason we choose this kind of mutation is because previous study show that in general the trends of this type of prediction models are usually good, but the threshold values can poses some problems [54]. Therefore only modifying the threshold values during mutation can preserve the validity of a rule (keeping the form of the condition). For example, when mutation is done on the above rule -"Rule 19", and suppose the first condition (*WMC <= 22*) is chosen, then mutation can be done to change the value *22* to another value, such as 21 (22 decreased by 1). After this mutation, the gene "Rule 19" will become the following (Figure 4.17). The changed condition is shaded:

Rule 19 :	*Gene Name*
WMC <= *21* MCC > 17 DEPCC > 2	*First Condition Mutated*
-> class 0 [63.3%]	*Conclusion*

Figure 4.17 A Condition Mutated in Figure 4.16

In our algorithm, a database is needed to define the given domain of the metrics. It can be a table as well. Table 4.1 is an example of this kind of database, which is constructed from the domain of stability prediction metrics values. This table contains the Name, Value Type, Value Range, average value, and Mutation Steps of the metrics. When mutation is performed, it takes out a condition according to the *Mutation Probability*, checks the metric name in this condition, then randomly takes a *step value* corresponding to the metric name. The new threshold value is obtained by using this *step value* added to (or subtracted from) its original threshold value. Finally it uses the

new value, substituting the original value, to get a mutated condition. After putting this mutated condition back to the rule, the mutated gene (a new rule) is produced. This table can be modified to satisfy different domain applications accordingly.

Table 4.1 The Metrics Database and Values

	NAME	TYPE	RANGE	AVERAGE	STEP
1	CHM	I	3—400	180	1,5,10
2	coh	R	0—1	0.5	0.05,0.05,0.1
3	COM	R	0—20	6	1,1,1
4	COMI	R	0—20	20	1,1,2
5	CUB	I	0—100	41	1,2,3
6	CUBF	I	1—100	60	1,2,3
7	DEPCC	R	0—490	280	1,5,10
8	DIT	I	1—20	5	1,1,1
9	LCOMB	R	0—3000	1300	1,10,20
10	MCC	R	0—490	280	1,5,10
11	MDS	I	0—400	170	1,5,10
12	NPA	I	0—100	40	1,2,3
13	NOC	I	1—50	18	1,2,4
14	NOCONT	I	1—3	1	1,1,1
15	NOM	I	1—1000	170	1,5,10
16	NON	I	1—20	5	1,1,1
17	NOP	I	1—50	17	1,1,2
18	NPPM	I	0—100	40	1,2,3
19	OCMAIC	I	0—40	40	1,2,3
20	OMAEC	I	1—150	60	1,2,3
21	WMC	R	0—1745	850	1,10,20
22	WMCLOC	R	0—5675	2600	1,10,20

- **Mutation on Default Rules**

If the selected rule (gene) for mutation is a *default rule*, the mutation operator just changes the class value to the opposite value (1 to 0 or 0 to 1). For example, to mutate gene "Default class: 0", the mutated gene will be "Default class: 1".

In general the process of mutation happens as in the following:

1. Obtain two new chromosomes by crossover.
2. Use the Mutation Checking Procedure to decide if mutation will occur.
3. If mutation is decided on, first randomly choose one model (chromosome) from the two.
4. Then randomly choose a rule (gene) from the selected model.
5. Check the rule type.

6. If it is a default rule, change it's class value to the opposite (1 to 0 or 0 to 1) and then finish.
7. If it is a basic rule, first decide either to increase or decrease metric threshold value by random
8. Then randomly choose a *Step Value* from the metrics domain database according to the metrics type, add (or subtract) this figure to (from) the threshold value in the condition, then finish.

4.6 Fitness Function

For each chromosome, it is necessary to measure how well it is suited to its environment. This measurement is its fitness value. We use the fitness function to obtain each chromosome's fitness value, which is also dependent on the environment (training data).

In our algorithm, the correctness function is used as the fitness decision function. The general formula of the correctness function is as follows:

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

Here f represents a chromosome and $C(f)$ is its correctness value (the fitness value). The "k" represents the total number of possible predicted values. The "n_{ij}" is the number of training vectors with real class c_i and the predicted class as c_j (Table 4.2).

Table 4.2 The Confusion Matrix of a Decision Function

		Predicted Class			
		c_1	c_2	...	c_k
Real Class	c_1	n_{11}	n_{12}	...	n_{1k}
	c_2	n_{21}	n_{22}	...	n_{2k}
	...	...	...	...	...
	c_k	n_{k1}	n_{k2}	...	n_{kk}

In the domain of our study, the range of output class is 0 and 1. The above table and the fitness function are specified as follows (Table 4.3):

Table 4.3 Confusion Matrix and Fitness Function for this Study

		Predicted Classifier	
		0	1
Real Classifier	0	n_{11}	n_{12}
	1	n_{21}	n_{22}

$$C(f) = \frac{n_{11} + n_{22}}{n_{11} + n_{12} + n_{12} + n_{22}}$$

In the Table 4.3:

n_{11} indicates the number of classes where the real classifier is 0 and the predicted classifier is 0.

n_{22} indicates the number of classes where the real classifier is 1 and the predicted classifier is 1.

n_{12} indicates the number of classes where the real classifier is 0 and the predicted classifier is 1.

n_{21} indicates the number of classes where the real classifier is 1 and the predicted classifier is 0.

$n_{11} + n_{22}$ refers to the total number of correct predictions.

$n_{12} + n_{21}$ refers to the total number of incorrect predictions.

In general, the fitness value is obtained by the total number of correct predictions divided by the total number of predictions. The highest value is 1 (meaning all predictions are correct) and the lowest is 0 (meaning no predictions are correct).

In our implementation, a group of source data was chosen as the environment to test the fitness of the created generation. The data set has the structure like the following example (Table 4.4)

Table 4.4 Data Enviroment

	Class / Metrics	C70	C71	C72	C73	.....
1	coh	0.4	0	0.6	0	.....
2	LCOMB	0	0	0	0	.....
3	COM	0	0.66667	0.4	0	.....
4	COMI	0	1	0.2	0	.....
5	OCMAIC	4	2	5	1	.....
6	CUBF	22	2	14	0	.....
7	CUB	22	2	14	1	.....
8	OMAEC	1	3	1	0	.....
9	NOC	0	0	0	0	.....
10	NOP	1	1	1	2	.....
11	NON	4	0	0	0	.....
12	NOCONT	0	1	1	0	.....
13	DIT	2	4	3	1	.....
14	MDS	7	55	14	0	.....
15	CHM	10	60	18	3	.....
16	NOM	4	6	5	0	.....
17	NPA	0	0	2	0	.....
18	NPPM	3	6	5	0	.....
19	WMC	9	8	18	0	.....
20	MCC	9	6	16	0	.....
21	DEPCC	2	0	7	0	.....
22	WMC_LOC	135	29	124	0	.....
	Real Classifier	1	1	0	1	.....
	Predict Classifier					.....

In this table, the columns *C70, C71, C72*...etc. are the names of the source data sets chosen for fitness testing in order to help the evolution. The rows named as coh, LCOMB, COM, etc. are the metrics chosen for measurement. This database is related to Table 4.1 (Section 4.4.3). Each of the metrics in this database has a description in Table 4.1 about its value range and mutation steps.

In Table 4.4, The "Real Classifier" value obtained by simply comparing the evolution of a class interface among the major version of the software. If they are the same, the "Real Classifier" value is assigned to 1, which means stable. Otherwise 0 is assigned which means unstable.

The "Predict Classifier" values are generated by the prediction model. Then the fitness function takes all the "Predict Classifier" values and the "Real Classifier" values to

calculate the fitness value (of this data environment) as described in Table 4.3.

4.7 Elitism

Elitism is used in our algorithm to ensure the best model's (chromosome) survival. After producing all the chromosomes in the new generation, then our algorithm copies the chromosome(s) with the best fitness value from their parent generation, as the elitism theory requires.

Elitism passes the best chromosome(s) to the next generation. This will avoid the loss of the best chromosomes from the present generation. Before the elitism is implemented, the chromosomes of the generation are sorted according to their fitness values. So the fittest chromosome will be in the first position of the generation.

The number of elite chromosomes that pass to the new generation is set as an input parameter in our algorithm. There are two ways to determine this elitism parameter. The first way is by making it an integer, such as 1, 2, ...etc. which indicates the exact number of chromosomes to be directly copied to the next generation. Another way is to set it as a percentage, such as 3%, 5%...etc. This indicates that the top 3% or 5% percent of the whole chromosome will be copied to the new generation.

4.8 Control of Population Size

After the new offspring are produced, our GA ranks all of them according to their fitness value. Therefore, the best ones are at the front and the worst at the end. Because the elite chromosomes are continually copied from generation to generation, the population of the new generation is gradually increasing. Therefore, we set another input parameter called "population size", which controls the maximum population size of a generation. In our algorithm, after the maximum population size allowable for a generation is set (suppose to be n), all numbers ranked after n will be abandoned from the new generation. This is done to reduce the processing time as well GA theory

suggests that the bad chromosomes should not survive.

4.9 Ending Condition

Our combination algorithm determines which individuals should reproduce, which should mutate, and which should survive or die. It also decides how long the evolution should continue. Typically a genetic algorithm does not have an obvious stopping criterion. Therefore, we must tell our algorithm when to stop. There are several criteria that can be used to stop the evolution, such as the number-of-generations, goodness-of-best-solution, convergence-of-population, or a problem-specific criterion as the algorithm ending condition.

In our study, most of the time the number-of-generations is used as a stopping measure (ending condition). After the new generation is created and tested, the survivors will become another parent generation. The process of selection, crossover, mutation, fitness test, sort, elitism and population control from step *Section 4.4* to *Section 4.8* are repeated and repeated until the generation number reaches the pre-set maximum number in a GA application process. Then the genetic algorithm stops and the best chromosome can be obtained from this evolution process. This is the best combination prediction model.

Another ending condition we use is the fitness improvement test. Our algorithm monitors the best fitness value as well as other fitness values. If the best fitness does not improve for certain generations (in our algorithm it is set to be 20 generations), or all the chromosomes' fitness in the current generation are the same value, the algorithm will stop.

4.10 The Main Generational Loop in Combination Algorithm

Now we have the models encoding and all the GA operations for the models. It's time to make them work in sequence to produce new generations, which can be called as **generational loop**.

A run of the main generational loop in our algorithm consists of the fitness evaluation, sort, elitism, population control, roulette wheel selection, crossover and mutation. Each chromosome in the current generation is evaluated to determine how fit it is at solving the problem (such as if a model has a higher prediction rate). Our algorithm then probabilistically selects from the current generation based on their fitness to participate in the various genetic operations. The more fit a chromosome is, the better chance it has to be selected. After the evolution of many generations, a chromosome (combined model) that is the best in the given data environment can be generated.

The summary of our genetic algorithm procedure is presented below:

PSEUDO CODE

```
// initialize the population of the first generation of the chromosome
P_0 := getInitialPopulation (Prediction Model Files);

// evaluate fitness of all initial chromosomes of current generation
evaluate ( P_0 );
// start with an initial time
t := 0;

// test for termination criterion (time and fitness)
while not done do

   // move the best chromosome to the next generation directly depending on the elitism
   setting
   P_t+1 := elitism( P_t )
   // repeat [generation size/2] times
   repeat
   // select a pair of chromosomes from the current generation for offspring production
```

```
        Parents := selectparents ( P_0 );

        // Crossover the "genes" of selected parents depending on the Probability of Crossover
        Children: = crossover (Parents);

        // Mutate the "genes" of children depending on the Probability of Mutation
        Children: = mutation (Children);

        // evaluate offspring's new fitness
        evaluate (Children);

      // add the children to the new generation

      P_{t+1} := addToNewGeneration(Children)

    end repeat
      // select the survivors from the new generation depending on the generation size control

        P_t := survive ( P_t );

        // increase the time counter
        t := t + 1;
      od
  //return the best fitness chromosome of the final generation

  return( P_t )

  end GA.
```

4.11 Summary of this Chapter

The main purpose of our research is to find a new approach to obtain new models through the combination of existing models. We adopt the genetic algorithm (GA) as our algorithm in this approach. Our algorithm is designed specifically to recombine the rule-based prediction models.

In this chapter we introduced how the GA works for our purpose. The models encoding is the most important step in our algorithm. The rule-based prediction models are chromosomes of our algorithm. Each model consists of a set of rules and a classifier

value. The encoding will affect the evolution efficiency. After that we defined our genetic operators to produce combination models. Crossover and mutation operators are dependent on the encoding method. The fitness function produces the measurement of how well each model is in a certain data environment. Elitism will let our algorithm avoid losing the best result. In the next chapter we will validate our algorithm through implementation and experimentation.

Chapter 5

Implementation and Experimentation

In this chapter we will demonstrate how our combination algorithm is applied to classic rule-based prediction models. Our experiment was performed on a "semi-real" environment. We used a 10-fold cross validation technique to estimate the combination models' accuracy. In this technique, the whole data set is split into 10 subsets of equal size. A combination model is trained on the union of 9 subsets (called a training data set) and tested on the remaining subset (called a testing data set).

5.1 Experimental Tool: GA-CAMP

To validate our combination algorithm we implemented an experimental tool called GA-CAMP (Genetic Algorithm used as a Combination Algorithm for the Models for Prediction) using Java language. Java has many features that make it an effective platform for our study. It is an object-oriented development tool and this adds an element of convenience for future studies in this area because many classes can be reused and the algorithm modification is very flexible. The platform is independent from Java and this allows our experimental tool to be run from anywhere. Figure 5.1 shows the GA-CAMP interface:

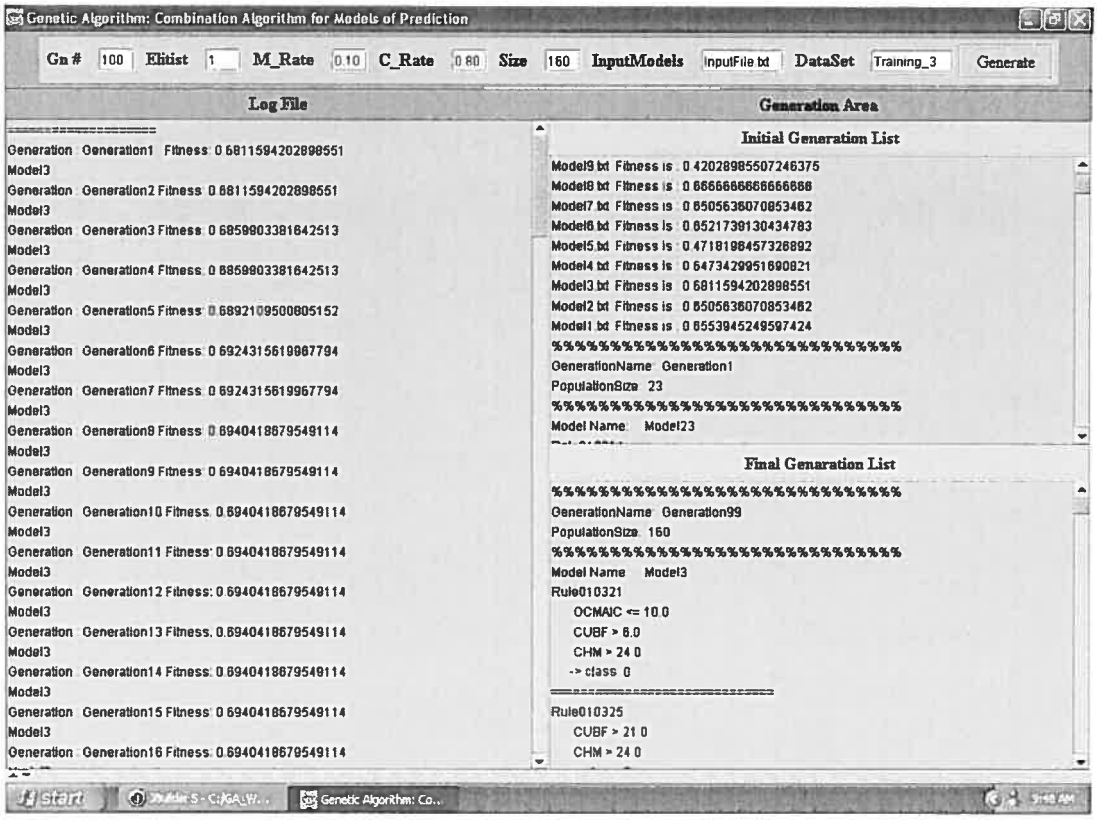


Figure 5.1 The Combination Algorithm Interface of GA-CAMP

GA-CAMP parses the decision-tree classifier files to obtain the classic rule-based prediction models. The decision-tree classifier file was produced by C4.5 algorithm (Quinlan, 1993) [51]. C4.5 is one of the empirical learning systems that constructs decision-tree classifiers. Figure 5.2 provides an example of this kind of file. GA-CAMP only takes out (exports) the model from this file and abandons other information. In Figure 5.2, the model is between the grey shadowed text, it starts from the string "Model Model 2" and ends at letter "$".

Model2.txt

```
C4.5 [release 8] rule generator Mon Jul 30 13:24:02 2001
-------------------------------

    Options:
    File stem <beanb92_her_comp_hid>

Read 390 cases (12 attributes) from beanb92_her_comp_hid

------------------
Processing tree 0

Final rules from tree 0:

Model Model 2 :

Rule 6:
       NON <= 0
       DIT <= 1
       MDS > 31
    ->  class 0  [45.3%]

Rule 4:
       NPPM <= 17
    ->  class 1  [95.6%]

Default class: 1

$

Evaluation on training data (390 items):

Rule  Size  Error  Used  Wrong            Advantage
----  ----  -----  ----  -----            ---------
   6     3  54.7%     4      1 (25.0%)      2 (3|1)  0
   4     1   4.4%   347     12 (3.5%)       0 (0|0)  1

Tested 390, errors 18 (4.6%)   <<


     (a)   (b) <-classified as
    ----  ----
     369     1 (a): class 1
      17     3 (b): class 0
```

Figure 5.2 An Example of Decision-Tree Classifier File Produced by C4.5

GA-CAMP has an input parameter named "InputModels", which is a file name we assign to let GA-CAMP know from where to get the initial models. This file contains a name list of all the decision-tree classifier files that are needed to do the combination.

After the combination, GA-CAMP provides the results in the "Final Generation List" area. It includes all the combination models in the final generation. The models are sorted according to their fitness values with the first one having the best fitness. The final results are saved in a file named "Results.txt".

Besides the "InputModels" parameter, GA-CAMP has another 5 important input parameters, which can be found in the GA-CAMP interface. These values may influence the final process efficiency as introduced in Chapter 4. They are:

1. **Gn# (Number of generations)**: this is the maximum number of generations that will be created. This is one of the combination algorithm stop conditions.
2. **Size (Population size)**: this is the maximum number of chromosomes in the current generation. A larger population size increases the amount of variation presented in the population at the expense of requiring more fitness function evaluations.
3. **Elitist**: this controls whether elitism is done or not, and how many of the best chromosomes will be transferred to the next generation directly.
4. **M_Rate (Mutation rate)**: this is the probability of the occurrence of mutation, the higher the mutation probability is, the more mutations will be done on the newborn chromosomes.
5. **C_Rate (Crossover Rate)**: this is similar to "M_Rate", it is the probability of the occurrence of crossover.

In the GA-CAMP interface parameter input area, there are three places showing the initial generation, final generation and general intermediate generation information; such as, the fitness values or the rule set names among other information. In the "Log

File" area (the box at the left), all of the messages produced during the running of the genetic algorithm will be displayed; such as, the generation number, the best fitness value and the best model name of each generation. The "Initial Generation List" area (the box at the upper right) displays the entire initial generation message in detail; such as, each model name and its fitness value. The "Final Generation List" area (the box at the bottom right) contains the result, the final generation of this running; such as, the best model name, the rule set and its fitness value.

GA-CAMP is the experimental tool used to evaluate our combination algorithm. After GA-CAMP is developed, we can conduct the experiment using this tool for the software prediction models. However, to do this we need an evaluation environment. This environment includes ***a set of prediction models*** and the ***training data sets*** from which the prediction models were built and the ***testing data sets*** used to do the evaluation for the resulting models.

Normally, the software prediction models are built from some kind of source data sets. However the source data sets from which the prediction models were built are rarely posted in the literature. Our algorithm is designed to be used with any kind of rule-based prediction model that can be collected from the posted literature. In order to verify the validity of our algorithm, not only should we obtain a set of combined prediction models through our algorithm, but also we should use the same testing data sets to evaluate the new models and the old ones. By using the same data to evaluate the old and new models, we can make a valid comparison and correct judgment of the results. Therefore, all the source data and input rule-based models make up our experiment environment.

Our experiment environment is a "semi-real" environment [55]. This is because the prediction models are simulated although the source data set is from real software systems: they are decision tree classifiers trained on independent software system data.

To imitate the heterogeneity of real-life prediction models, each model was trained on a different subset of metrics and on a different software system.

Although our algorithm can be applied to any kind of rule-based prediction model, the models used in our experiment are applied to predict the stability of the class interface between versions of software packages. This is to make the evaluation of software quality simple and accurate. Therefore, the principles and metrics of the software quality prediction model applied in our experiment are focused on stability. In order to understand the models in our experiment, a brief overview of the software stability concept and its measurement are provided in the next section.

5.2 Stability

The classic definition of the term stability is: "Not easily moved or changed." This definition can also be used in the software context. When we say a certain aspect of a class or a package (a group of classes) is stable, we mean that such an aspect is firm or hard to change. This characteristic can also be called "independence". An independent class is a class that does not depend upon anything else. The more stability a class has the more independent it is, and it is more reliable for reuse.

At present stability is the top consideration for all software design. When we design software, we strive to make it stable and aim at accomplishing system reusability. Indeed, this is the goal of modern software design. In the structure of an application, the stability impacts the relationships between packages because the packages are interrelated [44]. In fact, the way a stable model is built should guarantee its reusability.

- **Stability Measurement**

There are several methods used to measure the stability of a class. The measurement of stability depends on the application as well as the aspect of the class we need to

measure. For example, to measure the *positional* stability of a package, one way is to count the number of dependencies that enter and leave that package [44]. These measurements include the following metrics:

- *Ca* (Afferent Couplings): The number of classes outside this package that depend upon the classes within this package.
- *Ce* (Efferent Couplings): The number of classes inside this package that depend upon the classes outside this package.
- *I* (Instability): $(Ce \div (Ca+Ce))$: This metric has the range [0,1]. I=0 indicates a maximally stable package. I=1 indicates a maximally unstable package.

The *Ca* and *Ce* are the metrics used to calculate the positional stability of a package. These measurements are appropriate only for certain applications. For other applications (such as interface stability), there are different metrics that can be used to measure the stability of a class.

In our experiments we chose the models that predict the *interface stability* of Java classes. This is because that attribute is easy to measure and we can ensure the accuracy of the prediction model. The experiment looked at consecutive major versions of the same software to measure the stability. The definition is:

- *If a class* x_i *public interface of the* j^{th} *version is included in the public interface of the* $(j+1)^{th}$ *version, this class is stable (class 1).*
- *otherwise, it is unstable (class 0).*

The characteristic of stability is relatively less difficult to collect than others; such as, defect data or maintenance effort. It can be obtained by simply comparing the evolution of a class interface among the major version of the software and the result is easy and accurate to validate.

Our experiment started from the source data sets collected. Then a set of interface stability prediction models was extracted from the source date sets. After that we

applied our algorithm to combine the original models to get combination models. Finally we evaluated our results. In the following section we describe the experimental process and the results in detail.

5.3 Source Data Sets and Models Extract

In our experiment, we selected 11 Java systems that have at least two major versions. The size of the initial versions of the 11 systems, in the number of classes, is given in Table5.1. The metrics used as attributes in our experiment are extracted from these 11 systems. Nine systems, except for Jedit and Jetty, were used to "create" our prediction model (the original model) for our experiment. The remaining 2 systems, the Jedit (system #6) and Jetty (system #7), were selected for training the combination models and testing the combination results.

Table 5.1: The Software Systems Used to Train and to Combine the Models.

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

Twenty-two structural software metrics were extracted from these 11 software systems using the ACCESS tool of the Discover© environment. Discover© provides a powerful tool for the source code analysis, as well as navigation and query capabilities of existing software source code structure. It allows software developers to quickly find their way through code and to quickly understand a target software system. It supports many programming languages on various operating systems. Discover© is a parsing-based system that collects information about the relationships between language structures. More information about Discover© can be found at the web site: http://www.mks.com/products/discover/developer.shtml.

Table 5.2 provides the definitions of all 22 metrics extracted from the above 11 software systems. This table was also introduced in Chapter 2. We are presenting it again because our experiment models are constructed with these metrics. The 22 structural software metrics belong to one of the four categories of coupling, cohesion, inheritance, or complexity, and constitute a union of metrics used in different theoretical models [17, 7, 58, 12].

All these metrics were considered as independent parameters that have impact on the software stability.

Table 5.2 The 22 Software Metrics Used as Attributes in Our Experiments

Metrics		Description
Cohesion metrics		
1	LCOM	lack of cohesion Methods
2	COH	cohesion
3	COM	cohesion metric
4	COMI	cohesion metric inverse
Coupling metrics		
5	OCMAIC	other class method attribute import coupling
6	OCMAEC	other class method attribute export coupling
7	CUB	number of classes used by a class
8	CUBF	number of classes used by a memb. funct.

Inheritance metrics		
9	NOC	number of children
10	NOP	number of parents
11	NON	number of nested classes
12	NOCONT	number of containing classes
13	DIT	depth of inheritance
14	MDS	message domain size
15	CHM	class hierarchy metric
Size complexity metrics		
16	NOM	number of methods
17	WMC	weighted methods per class
18	WMCLOC	LOC weighted methods per class
19	MCC	McCabe's complexity weighted meth. per cl.
20	DEPCC	operation access metric
21	NPPM	number of public and protected meth. in a cl.
22	NPA	number of public attributes

After the 22 metrics for stability were extracted, the next step in our experiment was to build the prediction models from the 9 systems. The prediction models that are used in our experiment are generated by C4.5 [51] - a representative machine learning algorithm.

First, we started with enumerating the requirements for a classification task to be performable by C4.5. In our case, one classification task might be: "classify this class as interface stable or instable".

In order for C4.5 to work well, the following requirements should be applied [52]:

1. *Attribute-value description*: All information about one class should be expressible in terms of a fixed collection of attributes. In our experiment, all of them are given in Table 4.1.
2. *Predefined classes*: The categories to which classes will be assigned should be defined beforehand. When we are predicting the interface stability of a class in our experiment, we defined ours as being "1" for "stable" and "0" for "unstable".

3. *Discrete classes*: Classes should be sharply delineated. A case either belongs or does not belong to a certain class and there should be far more cases than classes.
4. *Sufficient data*: Sufficient cases should be available, as we don't want to leave much room for mere coincidences.
5. *Logical classification models*: The predictive models provided by C4.5 take the form of decision trees or production rules only.

Second, we should have a training set - a list of all the metric values assigned and their classification to every class (see Table 4.4). Then we provide C4.5 with the training set. C4.5 generates a classifier in the form of a decision tree where a leaf is a category and each no-leaf node is a test on one attribute value. The tree is used to classify a class by carrying out the test as indicated by the branches of the tree and moving through the tree from the root until a leaf is encountered. The tree is created as follows:

IF *all cases are of the same category* THEN

1. *create a leaf and label it with the name of this category.*

ELSE

2. *Select an attribute*
3. *Select a test based on this attribute*
4. *Divide the training set into subsets, each associated with one possible value of the tested attribute.*
5. *Apply the same procedure (Staring at the IF-statement) with each subset.*

END

After the decision tree is created, it is simplified by C4.5 with the aim of making it more comprehensible without compromising its accuracy. This step is referred to as pruning.

More details on how pruning is performed can be found in J. Quinlan's "C4.5: Programs for Machine Learning" [51]. Finally C4.5 converts the decision tree into a set of production rules or rule sets (see Figure 5.3).

The 23 initial models in our experiment were created from the 9 systems (see Table5.1 except Jedit and Jetty) in the following way:

- First we formed 15 subsets of the 22 software metrics by combing two, three, or four of the metrics categories in all the possible ways, and created *15X9=135* data sets.
- Then we trained a decision tree classifier on each data set using the C4.5 algorithm. We retained 23 decision trees by eliminating constant classifiers and classifiers with training errors of more than 10%.

Figure 5.2 is one of the 23 prediction models. All of the 23 prediction models used in our experiment are listed in Appendix A. In our experiment, the 23 classic rule-based models are the interface stability prediction models.

```
Model Model10 :

Rule 1:
      OMAEC <= 0
      -> class 1  [75.8%]

Rule 3:
      DIT > 1
      -> class 1  [50.0%]

Rule 2:
      OMAEC > 0
      DIT <= 1
      -> class 0  [87.1%]

Default class: 0
```

Figure 5.3 A Rule Set of a Decision Tree Created by C4.5

5.4 Experiment Settings

After we obtained the original models – 23 classic rule-based interface stability prediction models, we conducted our experiment in the two remaining systems in Table 5.1 - the Jedit and Jetty systems. We created a data set D_n that contains 690 data vectors (see Table 4.4 for the details of D_n) using the classes in these two systems as our data environment. The data environment was a database with 690 classes; each was named in sequence as "C1", "C2", "C3" … "C690". Each record has 22 metric values and a real classifier value.

To accurately estimate the correctness of the trained classifiers, we used a 10-fold cross-validation technique to evaluate our algorithm. Through this technique, the data set D_n (690 data vectors) is randomly split into 10 groups (subsets) of equal size (69 points each). The union of 9 subsets of source data (*69X9=621* points) is chosen as the training environment and is called the training data. Therefore, we have 10 different training datasets. During the evolution process, the training data was used to obtain the fitness values for each generation.

When the training data was selected, the remaining subset (69 data records) was used as the testing environment, also called the testing data. Therefore, we also have 10 different testing environments. Each training data set was paired with one testing data set accordingly, and both of them together made up an experiment environment. Therefore, we had 10 different experiment environments. Since both the training and testing data are a mixture of data from multiple systems, the cross-validity of obtained results is increased. In order to ensure reliability, our algorithm was applied to each environment. That is to say, we had to do 10 repetitions of our algorithm application.

The 10 subsets used for training were saved in 10 database files. The same procedure

occurred for the testing data sets. Table 5.3 shows the names of the training datasets and the testing datasets for each of the 10 repetitions.

Figure 5.3 The 10 Repetitions of Experiment Data Environments

Repetitions	Training Dataset	Testing Dataset
1st experiment	Training_1	Testing_1
2nd experiment	Training_2	Testing_2
3rd experiment	Training_3	Testing_3
4th experiment	Training_4	Testing_4
5th experiment	Training_5	Testing_5
6th experiment	Training_6	Testing_6
7th experiment	Training_7	Testing_7
8th experiment	Training_8	Testing_8
9th experiment	Training_9	Testing_9
10th experiment	Training_10	Testing_10

Because there is some random performance during the running of our algorithm, the same training data might lead to different results. In order to obtain a reliable result, we performed 6 iterations in each training experiment dataset. That is to say, for each algorithm application on the same training data environment, we ran it 6 times with different parameters, such as different mutation probabilities. The model with the best fitness value from the 6 was taken as the final one from this training environment. For the 10 training data sets, we were able to get 10 combination models, and each of them had the best fitness value in its training data.

Our genetic algorithm parameters (See section 5.1) needed to be set before the implementation of the algorithm. The elitism strategy was applied in the experiment as well: in each new generation, the majority of the population from the previous one was replaced, except for a small number N_e of the fittest chromosomes. In order to have a reasonable execution time, the number of total generations T was set to 100 and the maximum number of chromosomes (S) in a generation was set to 160. The values of

p_c (crossover probability), p_m (mutation probability) varied with the number of iteration (i). Table 5.4 indicates the actual parameter values used in the experiments.

Table 5.4 GA-CAMP Parameters

Iteration Number (i)	1st	2nd	3rd	4th	5th	6th
Number of Generations (T)	*100*	*100*	*100*	*100*	*100*	*100*
Elitist(N_e)*	*1*	*1*	*0.1*	*1*	*1*	*0.1*
Maximum Population Size(S)	*160*	*160*	*160*	*160*	*160*	*160*
Mutation Rate (p_m)	*0.05*	*0.05*	*0.05*	*0.10*	*0.10*	*0.10*
Crossover Rate(p_c)	*0.80*	*0.80*	*0.80*	*0.80*	*0.80*	*0.80*

** For Elitist, 1 means taking 1 chromosome; while 0.1 means 10% of the generation.*

5.5 Case Studies

In this section we describe how the results were obtained from our experiment. In order to have better results, the application process described in the previous section was repeated 6 times for each different training environment by setting different parameters. The model with the best fitness value was selected to be the final combination from this experiment.

In the following two sub-sections two of our experiments are demonstrated. All of the 10 experiment results can be found in Appendix B. The following presents the parameters set constant in all of the GA-CAMP experiments:

- **Number of Generations**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover Probability**: *0.8*

Input Models: 23 interface stability prediction models in classic rule-based models.

5.5.1 Case Study 1

Our first experiment is performed on the data set: *Training_1* and *Testing_1*. The results of all 6 iterations with the different values for the "*Mutation Probability*" and "*Elitist*" are shown below.

Table 5.5 The Results of the First Experiment

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.69%*
2nd	*5%*	*1*	*71.17%*
3rd	*5%*	*10%*	*70.37%*
4th	*10%*	*1*	*70.69%*
5th	*10%*	*1*	*73.10%*
6th	*10%*	*10%*	*70.69%*
Fitness value of combination model in training dataset: *73.10%*			
Fitness value of combination model in testing dataset: *72.10%*			
Best fitness value of original models in training: *69.08%*			
Best fitness value of original models in testing: *68.23%*			

In Table 5.5 the best fitness value of the original model and combination model from this experiment environment are listed. The combination model we obtained is:

```
Rule 011114:
     coh > 0.083735
     COMI > 0.16667
     COMI <= 2.875
     OCMAIC > 12.0
         -> class 1  [75.8%]

Rule 010422:
     DEPCC <= 18.0
         -> class 1  [86.9%]

Rule 010308
     OMAEC > 6.0
     DIT > 1.0
         -> class 1  [71.8%]

Rule 011106
     coh > 0.083735
     OCMAIC <= 13.0
         -> class 0  [93.5%]

Rule 011113
     coh > 0.133735
     COMI <= 0.16667
         -> class 1  [93.0%]

Default class:0
Fitness Value:0.7310789049919485
```

5.5.2 Case Study 2

Our second experiment is similar to the first one. It was performed on the data set: *Training_2 and Testting_2*. The results of all 6 iterations with the different values for the "*Mutation Probability*" and "*Elitist*" are shown below.

Table 5.6 The Results of the Second Experiment

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.37%*
2nd	*5%*	*1*	*68.92%*
3rd	*5%*	*10%*	*70.37%*
4th	*10%*	*1*	*72.30%*
5th	*10%*	*1*	*70.69%*
6th	*10%*	*10%*	*70.37%*
Fitness value of combination model in training dataset: *72.30%*			
Fitness value of combination model in testing dataset: *70.30%*			
Best fitness value of original models in training: *69.08%*			
Best fitness value of original models in testing: *66.23%*			

In Table 5.6 the best fitness value of the original model and combination model from this experiment environment are listed. The combination model we obtained is:

```
Rule Name: Rule011114
     coh > 0.033735
     COMI > 0.16667
     COMI <= 0.875
     OCMAIC > 10.0
         -> class 0  [75.8%]

Rule Name: Rule010325
     CUBF > 21.0
     CHM > 37.0
         -> class 0  [91.7%]

Rule Name: Rule010317
     NOP > 11.0
         -> class 1  [83.3%]

Rule Name: Rule011106
     coh > 0.083735
     OCMAIC <= 13.0
         -> class 1  [93.5%]

Rule Name: Rule011113
     coh > 0.033735
     COMI <= 0.16667
         -> class 1  [93.0%]

Default class:1
Fitness Value:0.7230273752012882
```

5.5.3 Case Study Summary

Our experiment was conducted on a data set of 690 Java classes and 23 interface stability prediction models using the 10-fold cross-validation technique. The 23 prediction models were trained on nine of the ten subsets and a combination model with the best fitness value in this environment was obtained. Then we tested this model on the remaining one subset (called the test data set) to verify the result. This experimental procedure was repeated on all of the 10 training environments. For the 23 input models, we also tested their fitness values for each training environment and testing data. We took the best fitness value from the input models, and compared them with the best ones from the obtained models. Since we had 10 experiment environments, we had 10 pairs of fitness values. Table 5.7 summarizes all of our 10 experiment results.

Table 5.7 Fitness Values from Training and Testing Data

	Training Data		Testing Data	
	f_{best} (%)	f_{gen} (%)	f_{best} (%)	f_{gen} (%)
1	69.08	73.10	68.23	72.10
2	69.08	72.30	66.23	70.30
3	68.11	71.65	69.10	71.65
4	70.37	71.81	70.22	73.25
5	68.59	70.85	65.43	70.20
6	68.76	72.62	64.51	70.11
7	68.27	70.85	66.54	69.31
8	68.72	70.04	67.12	68.12
9	70.04	72.30	69.12	72.30
10	68.43	71.33	68.23	70.37

* *f_{gen} refers to the fitness values of combination models; f_{best} refers to best fitness values of the input models.*

5.6. Results

Through the application of this algorithm we finally obtained 10 combination models with the best fitness values from the 10 repetitions of our experiments. The next step required us to find out if the new models had better fitness values than those of the input models.

For each combination model, there is two fitness values: one from its training data and another from the test data. In order to get reliable results, we used the same training and testing data to get the fitness values of the input models. Since we had 23 original input models, we got 23 pairs of fitness values from each training and testing data set. We only selected the original model with the best training fitness value to compare with the final combination models.

Table 5.7 lists the fitness values we got from the training datasets and testing datasets. We decided to use statistical tests to compare their means. Obviously, this comparison is significant only if the data are from same environment. Therefore, the fitness values from the training data can only be compared with those from the training data, and the values from the testing data can only be compared with those from the testing data.

What we wanted to find out was whether the mean of the combination models' fitness value is significantly higher than the original model. If it is, then we can conclude that the GA method can really help in obtaining better models from existing models. Furthermore, because we used the fitness values from multiple systems` data , the model was cross-valid. That is to say, our combination algorithm can be used as an evolutionary approach to build a prediction model and the new model is more generally applicable.

We present the fitness values in a graphic form (Figure 5.4 and 5.5), where the combination models' fitness values are clearly seen to be higher in both the training and testing data sets. Therefore, the test results prove that our approach of combining exiting models can yield significantly better results than using individual models.

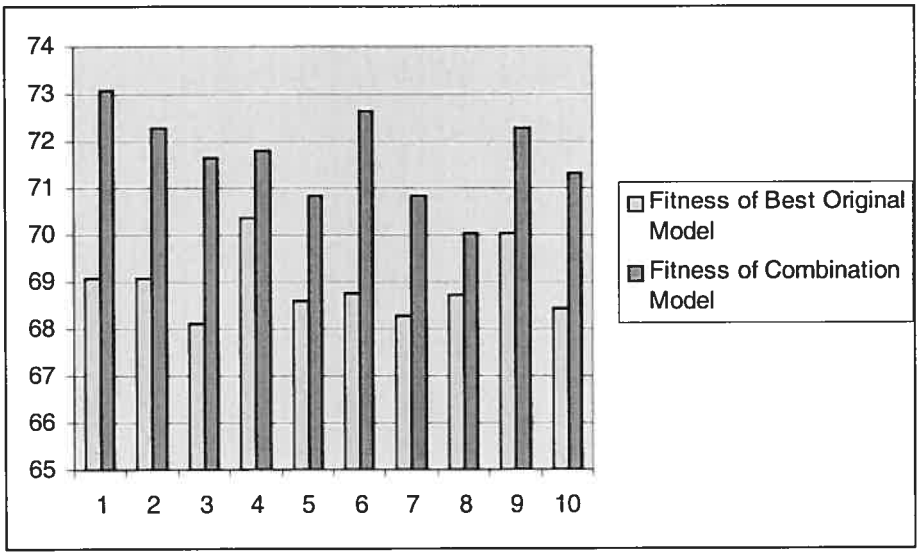


Figure 5.4 The Original and Combination Models' Fitness Values on the Training Data

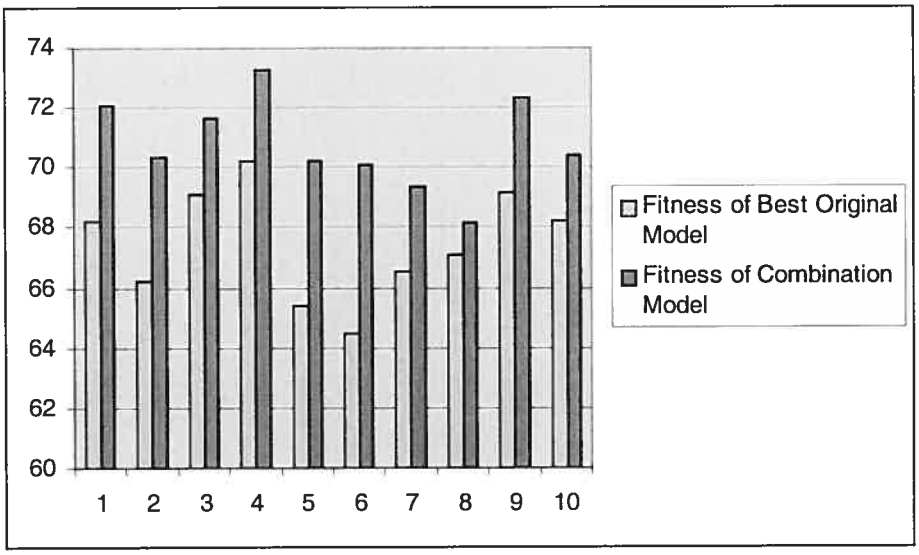


Figure 5.5 The Original and Combination Models' Fitness Values on the Testing Data

The statistical results in the training data and testing data are shown in Table 5.8.

Table 5.8 Experimental Results

		Fitness F(*f*)*
Training	f_{best}	*68.95 (0.54)*
	f_{gen}	*71.69 (0.89)*
Testing	f_{best}	*67.47 (3.28)*
	f_{gen}	*70.77 (2.37)*

* *The mean(standard deviation) percentage values of the correctness.*

From Table 5.8, we can see the fitness rates are relatively low, which indicates that the chosen problem of predicting software quality is a difficult problem. The results from the training data are strongly in favor of our proposal because the combination models' fitness values are significantly higher than the best fitness values of the original input models. However, the results from the testing data are somewhat unclear. The large standard deviations decreased our confidence level. However, we believe the large variation was caused by the outliers (extreme values), whose effects were enlarged in a small data set (the test data had only 69 data vectors).

In general, although we are aware of the limitations of this experiment, we found that it simulated reasonably well a realistic situation and yielded some interesting results. We strongly believe that if we use more numerous and real models on cleaner, less ambiguous data, the improvement will be even more significant. In particular, the results show that genetic algorithms can be used to improve the prediction ability of existing classic rule-based models.

5.7 Summary of this Chapter

In this chapter, we explained the implementation (GA-CAMP) of our combination algorithm for classic rule-based prediction models, and how we applied GA-CAMP in a "semi-real" environment to evaluate our combination algorithm with the 10-fold cross validation technique. This "semi-real" environment consists of a real software system data set and a set of "simulated" prediction models built from it. Finally, we presented our experiment results and proposed our conclusion: a genetic algorithm can be used as an evolutionary approach for combining and improving software quality prediction models in a particular context.

Chapter 6

Conclusion and Future Work

In this chapter we will summarize the work done in this thesis and propose directions for further research that seem to be worthy to be explored in this area.

6.1 Summary

Predicting software quality in the early stage of the software lifecycle has been an area of interest for a long time. Software prediction models offer an interesting solution to this problem. Normally, the approach of building prediction models is either from historical data or from experts with specific heuristic knowledge which can only be applied to the specific context from which they were built. Unfortunately, we cannot build software quality prediction models for software organizations if they lack historical data. Meanwhile a lot of software quality prediction models have been proposed in the literature.

Our research has proposed an evolutionary approach by using a genetic algorithm for combining and adapting existing software quality predictive models from a particular context. It was also taken as an exploratory phase that offered proof to the concept of

combining existing models using genetic algorithm. The resulting model can be interpreted as a "meta-model" that selects the best model for each given task. This notion corresponds well to the "real world" in which individual predictive models, coming from heterogeneous sources, are not universal, and depend largely on the underlying data. Our results show that the combination of models can perform better than individual models, even within a multiple systems context. On the basis of this study, some techniques (such as the method of model coding and the crossover operator) were improved by the students who continued this research. The results from this study were also confirmed by the continued study [55].

In our research, our contribution is:

- First, we proposed a new approach to develop cross-valid software quality prediction models. This method is especially beneficial for companies lacking in historical data.
- Second, our proposed approach can also be used to improve the efficiency and prediction ability of existing rule-based software prediction models.
- Third, it is believed, though more research is needed, that using genetic algorithms as a combination technique for improving the efficiency of rule-based models of multiple contexts is viable.
- Fourth, it is shown that this approach works well for interface stability prediction in real software systems.
- Last but not least, we developed a platform-independent tool GA-CAMP, which is completely object-oriented and self-contained. This will greatly benefit future researchers in this area by reusing classes and modules.

From our research, we argue that local search methods like genetic algorithms can be appropriate for hard problem solutions.

6.2 Future Work

Future work could focus on the following aspects. To show the universality of our technique, we also intend to evaluate our method on the data coming from other domains where representative benchmarks exist. To repeat this experiment with more accurate and a larger data size could be beneficial. In this kind of experiment a better definition of stability or another quality factor needs to be made. This experiment could be done on more systems and the fitness function could be improved also.

The techniques, such as the models encoding, can also be improved for further work. In this thesis we implemented the genetic algorithm for the classic rule-based prediction models--a decision tree classifier in the linear representation. For example, the model encoding could be represented as a set of isothetic boxes, which is two-dimensional, coming from the decision tree output regions directly. In such encoding, the genetic algorithm operators will need to be modified to fit this kind of encoding in order to preserve consistency and completeness. Some students have been working on this direction [55].

Issues for future research include the evaluation of this approach on real models proposed in the literature and the comparison of our approach to other white-box techniques. We also suggest testing other local search algorithms (Tabu Search or Aimuloted anneoling) in this domain.

Bibliography

[1] Mauricio A. de Almeida, Hakim Lounis, Walcélio L. Melo, "An Investigation on the Use of Machine Learned Models for Estimating Correction Costs", IEEE Computer Society Washington, DC, USA, 473 – 476,1998

[2] F. Akiyama, "An Example of Software System Debugging," Information Processing, vol. 71, pp. 353-379, 1971.

[3] V. R. Bassili, L. Brain & W. Melo, "How Reuse Influences Productivity in Object-Oriented Systems". Communications of the ACM, Vol. 30, N. 10, pp104-114, 1996.

[4] V. R. Basili, L. Briand, and W.L. Melo, "A Validation of Object Oriented Design Metrics as Quality Indicators," IEEE Trans. Software Eng., 1996.

[5] L. Briand, W. Thomas, C. Hetmanski. "Modeling and Managing Risk Early in Software Development", IEEE International Conference on Software Engineering (ICSE), 1993, Baltimore, Maryland, USA

[6] L. Briand, W. Melo, C. Seaman, and V. Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization," in Proceedings of the 17th International Conference on Software Engineering, pp. 133-143, 1995.

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

[8] L. Briand, Ralf Kempkens, Kerstin Lünenbürger, Michael Ochs, Martin Verlage, "Modelling the Factors Driving the Quality of Meetings in the Software Development Process", European Software Cost Estimation and Measurement (ESCOM'99), England, 1999

[9] L. Briand, J. Wuest, S. Ikonomovski, and H. Lounis: "Investigating Quality Factors in ObjectOriented Designs: An Industrial Case Study". In Proceedings of the International Conference on Software Engineering, 1999.

[10] L. Briand, K. El Emam, B. Freimut, O. Laitenberger, "A Comprehensive Evaluation of Capture-Recapture Models for Estimating Software Defect Content ", IEEE Transactions on Software Engineering, Volume 26, No 6, June 2000.

[11] L. Briand, T. Langley, I. Wieczorek, "A replicated Assessment and Comparison of Common Software Cost Modeling Techniques", IEEE International Conference on Software Engineering (ICSE), 2000, Limerick, Ireland

[12] L. Briand, J. W¨ust, John W. Daly, and V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," Journal of Systems and Software, vol. 51, pp. 245–273, 2000.

[13] L. Briand, Jürgen Wüst, "Modeling Development Effort in Object-Oriented Systems Using Design Properties", IEEE Software Engineering, pp. 963-986, November 2001 (Vol. 27, No. 11)

[14] L. Briand, J. Wuest, H. Lounis, "Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs", Empirical Software Engineering: An International Journal, Vol. 6. No 1, 11-58,2001.

[15] L. Briand, J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems", Advances in Computers, 2002, Academic Press, updated Feb. 18, 2002

[16] L. Briand, W. Melo, J. Wuest ; "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects"; International Software Engineering Research Network (ISERN), 2000; ISERN-00-06 Version 2. IEEE Transactions on Software Engineering, 2002

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions of Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[18] S. Chulani, B. Boehm, and B. Steece. "Bayesian Analysis of Empirical Software Engineering Cost Models". IEEE Transaction on Software Engineerining, 25(4), July/August 1999.

[19] T. Compton, and C. Withrow, "Prediction and Control of Ada Software Defects," J. Systems and Software, vol. 12, pp. 199-207, 1990.

[20] Ebert, C. "Metrics for Identifying Critical Components in Software Projects," Handbook of Software Engineering and Knowledge Engineering. 2001.

[21] K. El-Emam and W. Melo, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," Technical Report, National Research Council of Canada, NRC/ERB 1064 1999.

[22] Falkenauer, E."Genetic Algorithms and Grouping Problems", John Wiley and Sons Ltd. England, 1998.

[23] N. Fenton and B.A. Kitchenham, "Validating Software Measures," J. Software Testing, Verification & Reliability, vol. 1, no. 2, pp. 27-42, 1991.

[24] N. Fenton and M. Neil: "A Critique of Software Defect Prediction Models". In IEEE Transactions on Software Engineering, 25(5):676-689, 1999.

[25] N. Fenton and Neil M, "Software Metrics and Risk", Proc 2nd European Software Measurement Conference (FESMA'99), TI-KVIV, Amsterdam, ISBN 90-76019-07-X, pp 39-55, 1999.

[26] N. Fenton and M. Neil. "Software Metrics: A Roadmap". In A. Finkelstein, editor, The Future of Software Engineering. ACM Press, New York, 2000.

[27] N. Fenton and Ohlsson N, "Quantitative Analysis of Faults and Failures in a Complex Software System", IEEE Transactions on Software Engineering, 26(8), 797-814, 2000.

[28] N. Fenton, Krause P, Neil M, "A Probabilistic Model for Software Defect Prediction", IEEE Transactions in Software Engineering, Sept 2001

[29] N. Fenton, Krause P, Neil M, "Software Measurement: Uncertainty and Causal Modelling", IEEE Software 10(4), 116-122, 2002

[30] Florac, William A. et al. "Software Quality Measurement: A Framework for Counting Problems and Defects". (CMU/SEI-92-TR22) . Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, September 1992.

[31] J.R. Gaffney, "Estimating the Number of Faults in Code," IEEE Trans. Software Eng., vol. 10, no. 4, 1984.

[32] Gibbs, W.W. "Software's Chronic Crisis". Scientific American (September 1994), 86-95.

[33] Gray, A.R. and MacDonell, S.G. (1997) "A Comparison of Techniques for Developing Predictive Models of Software Metrics." Information and Software Technology 39(6), pp425-437, 1997.

[34] M.H. Halstead, "Elements of Software Science." Elsevier, North-Holland, 1975.

[35] Hilburn, T. B. and Towhidnejad, M., "Software Quality: A Curriculum Postscript?", SIGCSE Bulletin March 2000.

[36] Holland, J., "Adaptation in Natural and Artificial Systems An Introductory Analysis with Applications to Biology, Control, and Artificial intelligence". MIT Press, 1992.

[37] Jones, Capers, "Software Metrics," IEEE Computer, pp. 98--100, September, 1994.

[38] Khaled El-Emam, "A Methodology for Validating Software Product Metrics", Technical Report, NRC/ERB-1076, June 2000. 39 pages. NRC 44142.

[39] Khoshgoftaar, T. and D.L. Lanning: A Neural Network Approach for Early Detection of Program Modules Having High Risk in the Maintenance Phase. J. Systems and Software, Vol. 29, pp. 85-91, 1995.

[40] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman, "An Evaluation of Some Design Metrics," Software Eng J., vol. 5, no. 1, pp. 50-58, 1990.

[41] M. Lipow, "Number of Faults per Line of Code," IEEE Trans. Software Eng., vol. 8, no. 4, pp. 437-439, 1982.

[42] Luger, G.F. and Stubblefield, W.A. "Artificial intelligence." In MacMillan Encyclopedia of Computer Science. New York, NY: MacMillan (1991).

[43] Y. Mao, H. A. Sahraoui and H. Lounis, "Reusability Hypothesis Verification Using Machine Learning Techniques: A Case Study", Proc. of IEEE Automated Software Engineering Conference, 1998.

[44] R.C. Martin, "Stability", C++ Report, Feb. 1997.

[45] T.J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., vol. 2, no. 4, pp. 308 - 320, 1976.

[46] T. Mitchell, "Decision Tree Learning", in T. Mitchell, Machine Learning, The McGraw-Hill Companies, Inc, pp. 52-78, 1997.

[47] Musa, John D.; Iannino, Anthony; & Okumuto, Kazihra. "Software Reliability Measurement, Prediction, Application". New York, N.Y.:McGraw-Hill, 1987.

[48] Neil M, Fenton NE, Nielsen L, "Building large-scale Bayesian Networks", The Knowledge Engineering Review, 15(3), 257-284, 2000.

[49] N. Ohlsson and H. Alberg "Predicting Error-Prone Software Modules in Telephone Switches, IEEE Trans. Software Eng., vol. 22, no. 12, pp. 886-894, 1996.

[50] Paulk, Mark C.; Curtis, Bill; & Chrissis, Mary Beth; "Capability Maturity Model for Software" (CMU/SEI-91-TR-24, ADA 240603). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University,1991.

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

[52] Houari A. Sahraoui, Danielle Azar "Quality Estimation Models Optimization Using Genetic Algorithms: Case of Maintainability". In proc. of the European Software Measurement Conference (FESMA), 1999.

[53] Houari A. Sahraoui, Mounir Boukadoum, Hassan M. Chawiche, Gang Mail, Mohamed Serhani, "A fuzzy logic framework to improve the performance and interpretation of rule-based quality prediction models for object-oriented software". In proceedings of the 26th Computer Software and Applications Conference (COMPSAC), 2002

[54] H. A. Sahraoui, M. Boukadoum, H. Lounis, F. Ethève, "Predicting Class Libraries Interface Evolution: an investigation into machine learning approaches", In Proc. of 7th Asia-Pacific Software Engineering Conference, 2000.

[55] Salah Bouktif, Balàzs Kègl, Houari Sahraoui, "Combining Software Quality Predictive Models: An Evolutionary Approach". In proc. of IEEE International Conference on Software Maintenance (ICSM), 2002.

[56] Ian Sommerville, "Software Engineering", Addison Wesley.6th Edition, 2001.

[57] P. Winston, "Learning by Building Identification Trees", in P. Winston, Artificial Intelligence, Addison-Wesley Publishing Company, pp. 423-442, 1992.

[58] Wolfgang and Banzhaf. Genetic Programming, "An Introduction: On the Automatic Evolution of Computer Programs and Its Applications". Book, Morgan Kaufman Publisher, ISBN 1-55860-510-X, First Edition, 1997.

[59] H. Zuse, "A Framework of Software Measurement", Walter de Gruyter, 1998.

Appendix A Classic Rule-based Prediction Models for Stability

The 1st Model: Model1

```
C4.5 [release 8] rule generator     Mon Jul 30 13:13:44 2001
-------------------------------

    Options:
      File stem <beanb92_coh_comp>

Read 390 cases (11 attributes) from beanb92_coh_comp

------------------
Processing tree 0

Final rules from tree 0:

Model Model1 :

Rule 7:
      LCOMB > 16
      NPPM <= 10
      ->  class 0  [50.0%]

Rule 2:
      LCOMB <= 16
      ->  class 1  [95.7%]

Default class: 1

$

Evaluation on training data (390 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
   7     2  50.0%     2      0 (0.0%)          2 (2|0)       0
   2     1   4.3%   351     12 (3.4%)          0 (0|0)       1

Tested 390, errors 18 (4.6%)   <<


        (a)   (b)  <-classified as
       ----  ----
        370        (a): class 1
         18     2  (b): class 0
```

The 2nd Model: Model2

```
C4.5 [release 8] rule generator     Mon Jul 30 13:24:02 2001
-------------------------------

    Options:
      File stem <beanb92_her_comp_hid>

Read 390 cases (12 attributes) from beanb92_her_comp_hid

------------------
Processing tree 0

Final rules from tree 0:

Model Model2 :

Rule 6:
      NON <= 0
      DIT <= 1
      MDS > 31
      ->  class 0  [45.3%]

Rule 4:
      NPPM <= 17
      ->  class 1  [95.6%]

Default class: 1

$

Evaluation on training data (390 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
   6     3  54.7%     4      1 (25.0%)          2 (3|1)       0
   4     1   4.4%   347     12 (3.5%)           0 (0|0)       1

Tested 390, errors 18 (4.6%)   <<


        (a)   (b)  <-classified as
       ----  ----
        369     1  (a): class 1
         17     3  (b): class 0
```

The 3rd Model: Model3

```
C4.5 [release 8] rule generator     Fri Jul 27 03:11:58 2001
-------------------------------

    Options:
      File stem <beanb94_her_coup>

Read 387 cases (11 attributes) from beanb94_her_coup

------------------
Processing tree 0

Final rules from tree 0:

Model Model3 :

Rule 21:
      OCMAIC <= 7
      CUBF > 6
      CHM > 24
      ->  class 0  [94.4%]

Rule 25:
      CUBF > 21
      CHM > 24
      ->  class 0  [91.7%]

Rule 17:
      NOP > 4
      ->  class 0  [83.3%]

Rule 13:
      CUBF > 8
      NOC > 2
      ->  class 0  [79.4%]

Rule 16:
      OMAEC > 1
      NOP > 2
      ->  class 0  [76.1%]

Rule 8:
      OMAEC > 6
      DIT > 1
      ->  class 0  [71.8%]

Rule 18:
      CUBF <= 6
      OMAEC <= 7
      NOP <= 1
      ->  class 1  [91.8%]
```

Continue in next page

Continued from last page

```
Rule 15:
      OMAEC <= 1
      NOP <= 4
      CHM <= 24
      ->  class 1  [88.9%]

Rule 14:
      NOP <= 2
      CHM <= 24
      ->  class 1  [88.4%]

Default class: 1

$

Evaluation on training data (387 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
  21     3   5.6%    24      0 (0.0%)         11 (11|0)      0
  25     2   8.3%    14      0 (0.0%)          6 (6|0)       0
  17     1  16.7%     6      1 (16.7%)         5 (5|0)       0
  13     2  20.6%     3      0 (0.0%)          3 (3|0)       0
  16     2  23.9%    10      3 (30.0%)         4 (6|2)       0
   8     2  28.2%     9      3 (33.3%)         3 (6|3)       0
  18     3   8.2%   130      8 (6.2%)          0 (0|0)       1
  15     3  11.1%   137     14 (10.2%)         0 (0|0)       1
  14     2  11.6%    47      6 (12.8%)         0 (0|0)       1

Tested 387, errors 36 (9.3%)   <<


        (a)   (b)   <-classified as
       ----  ----
        292     7   (a): class 1
         29    59   (b): class 0
```

The 4th Model: Model4

```
C4.5 [release 8] rule generator     Fri Jul 27 03:12:38 2001
-------------------------------

    Options:
      File stem <beanb94_coh_comp_hid>

Read 387 cases (11 attributes) from beanb94_coh_comp_hid

------------------
Processing tree 0

Final rules from tree 0:

Model Model4 :

Rule 12:
      COM <= 0
      NOM > 5
      NOM <= 6
      NPPM > 4
      DEPCC > 0
      ->  class 0  [82.0%]

Rule 23:
      NPPM > 16
      DEPCC > 2
      ->  class 0  [79.6%]

Rule 19:
      WMC <= 22
      MCC > 17
      DEPCC > 2
      ->  class 0  [63.3%]

Rule 22:
      DEPCC <= 2
      ->  class 1  [86.9%]

Rule 13:
      MCC <= 17
      ->  class 1  [86.3%]

Default class: 1

$
```

Continue in next page

Continued from last page

```
Evaluation on training data (387 items):
Rule  Size  Error  Used  Wrong              Advantage
----  ----  -----  ----  -----              ---------
  12     5  18.0%     7      0 (0.0%)        7 (7|0)      0
  23     2  20.4%    35      5 (14.3%)      24 (29|5)     0
  19     3  36.7%    21      6 (28.6%)       9 (15|6)     0
  22     1  13.1%   244     26 (10.7%)       0 (0|0)      1
  13     1  13.7%    47      5 (10.6%)       0 (0|0)      1

Tested 387, errors 47 (12.1%)   <<


       (a)   (b)   <-classified as
      ----  ----
       288    11   (a): class 1
        36    52   (b): class 0
```

The 5th Model: Model5

```
C4.5 [release 8] rule generator     Fri Jul 27 03:14:01 2001
-------------------------------------------------------------

    Options:
      File stem <free100_coh_coup>

Read 49 cases (10 attributes) from free100_coh_coup

-------------------------------------------------------------
Processing tree 0

Final rules from tree 0:

Model Model5 :

Rule 5:
      CUBF > 7
      ➔  class 1  [95.2%]

Rule 4:
      NOP > 2
      ➔  class 1  [89.1%]

Rule 1:
      CUB <= 2
      NOP <= 2
      ➔  class 0  [45.3%]

Default class: 1

$

Evaluation on training data (49 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   5     1   4.8%    28      0 (0.0%)          0 (0|0)      1
   4     1  10.9%    10      0 (0.0%)          0 (0|0)      1
   1     2  54.7%     4      1 (25.0%)         2 (3|1)      0

Tested 49, errors 4 (8.2%)   <<


        (a)   (b)  <-classified as
       ----  ----
         42     1  (a): class 1
          3     3  (b): class 0
```

The 6th Model: Model6

```
C4.5 [release 8] rule generator     Fri Jul 27 03:16:55 2001
-------------------------------------------------------------

    Options:
      File stem <free100_her_comp>

Read 49 cases (12 attributes) from free100_her_comp

-------------------------------------------------------------
Processing tree 0

Final rules from tree 0:

Model Model6 :

Rule 3:
      WMCLOC > 42
      ➔  class 1  [91.7%]

Rule 1:
      NOCONT <= 0
      ➔  class 1  [87.1%]

Rule 2:
      NOCONT > 0
      WMCLOC <= 42
      ➔  class 0  [44.1%]

Default class: 1

$

Evaluation on training data (49 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   3     1   8.3%    31      1 (3.2%)          0 (0|0)      1
   1     1  12.9%    10      0 (0.0%)          0 (0|0)      1
   2     2  55.9%     8      3 (37.5%)         2 (5|3)      0

Tested 49, errors 4 (8.2%)   <<


        (a)   (b)   <-classified as
       ----  ----
         40     3   (a): class 1
          1     5   (b): class 0
```

The 7th Model: Model7

```
C4.5 [release 8] rule generator     Fri Jul 27 03:18:16 2001
-------------------------------------------------------------

    Options:
      File stem <free14_coh_comp>

Read 69 cases (11 attributes) from free14_coh_comp

-------------------------------------------------------------
Processing tree 0

Final rules from tree 0:

Model Model7 :

Rule 5:
      NOM <= 9
      NPA > 1
      DEPCC <= 7
      ➔  class 0  [93.0%]

Rule 4:
      NPA <= 1
      ➔  class 1  [83.3%]

Default class: 1

$

Evaluation on training data (69 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   5     3   7.0%    19      0 (0.0%)         19 (19|0)      0
   4     1  16.7%    43      5 (11.6%)         0 (0|0)       1

Tested 69, errors 7 (10.1%)   <<


        (a)  (b)  <-classified as
       ---- ----
        ➔         (a): class 1
         7   19   (b): class 0
```

The 8th Model: Model8

```
C4.5 [release 8] rule generator     Fri Jul 27 03:19:02 2001
-------------------------------------------------------------

    Options:
      File stem <free14_coup_her>

Read 69 cases (11 attributes) from free14_coup_her

-------------------------------------------------------------
Processing tree 0

Final rules from tree 0:

Model Model8 :

Rule 4:
      CUBF > 12
      ➔  class 0  [88.5%]

Rule 3:
      OCMAIC > 1
      NOP > 1
      ➔  class 1  [93.3%]

Rule 2:
      CUBF <= 12
      ➔  class 1  [84.7%]

Default class: 1

$

Evaluation on training data (69 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   4     1  11.5%    22      1 (4.5%)         20 (21|1)      0
   3     2   6.7%    19      0 (0.0%)          0 (0|0)       1
   2     1  15.3%    28      5 (17.9%)         0 (0|0)       1

Tested 69, errors 6 (8.7%)   <<


        (a)   (b)   <-classified as
       ----  ----
         42     1   (a): class 1
          5    21   (b): class 0
```

The 9th Model: Model9

```
C4.5 [release 8] rule generator     Fri Jul 27 03:19:56 2001
-------------------------------------------------------------

    Options:
      File stem <javamapper_coh_her_comp>

Read 17 cases (18 attributes) from javamapper_coh_her_comp

-------------------------------------------------------------
Processing tree 0

Final rules from tree 0:

Model Model9 :

Rule 1:
      coh <= 0.11448
      NOCONT <= 0
      ➔  class 1  [79.4%]

Rule 3:
      NOCONT > 0
      ➔  class 0  [75.8%]

Rule 2:
      coh > 0.11448
      ➔  class 0  [70.0%]

Default class: 0

$

Evaluation on training data (17 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   1     2  20.6%     6      0 (0.0%)          6 (6|0)      1
   3     1  24.2%     5      0 (0.0%)          0 (0|0)      0
   2     1  30.0%     6      1 (16.7%)         0 (0|0)      0

Tested 17, errors 1 (5.9%)   <<


        (a)   (b)  <-classified as
       ----  ----
          6     1  (a): class 1
              ➔  (b): class 0
```

The 10th Model: Model10

```
C4.5 [release 8] rule generator     Fri Jul 27 03:20:38 2001
-------------------------------------------------------------

    Options:
      File stem <javamapper_coup_her_comp>

Read 17 cases (18 attributes) from javamapper_coup_her_comp

-------------------------------------------------------------
Processing tree 0


Final rules from tree 0:

Model Model10 :

Rule 1:
      OMAEC <= 0
      ➔  class 1  [75.8%]

Rule 3:
      DIT > 1
      ➔  class 1  [50.0%]

Rule 2:
      OMAEC > 0
      DIT <= 1
      ➔  class 0  [87.1%]

Default class: 0

$

Evaluation on training data (17 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   1     1  24.2%     5      0 (0.0%)          5 (5|0)      1
   3     1  50.0%     2      0 (0.0%)          2 (2|0)      1
   2     2  12.9%    10      0 (0.0%)          0 (0|0)      0

Tested 17, errors 0 (0.0%)   <<


        (a)   (b)  <-classified as
       ----  ----
          ➔        (a): class 1
              ➔    (b): class 0
```

The 11th Model: Model11

```
C4.5 [release 8] rule generator     Tue Jul 31 01:28:34 2001
-------------------------------------------------------------

    Options:
      File stem <jigsaw205_coh_coup>

Read 868 cases (10 attributes) from jigsaw205_coh_coup

-------------------------------------------------------------
Processing tree 0

Final rules from tree 0:

Model Model11 :

Rule 14:
      coh > 0.033735
      COMI > 0.16667
      COMI <= 0.875
      OCMAIC > 10
      ➔  class 0  [75.8%]

Rule 3:
      coh <= 0.033735
      COM <= 0.15789
      OCMAIC > 2
      OCMAIC <= 4
      CUBF > 4
      CUBF <= 7
      ➔  class 0  [73.3%]

Rule 1:
      OCMAIC <= 2
      ➔  class 1  [97.1%]

Rule 6:
      coh > 0.033735
      OCMAIC <= 10
      ➔  class 1  [93.5%]

Rule 13:
      coh > 0.033735
      COMI <= 0.16667
      ➔  class 1  [93.0%]

Default class: 1

$
```

Continue in next page

Continued from last page

```
Evaluation on training data (868 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
  14     4  24.2%     5      0 (0.0%)          5 (5|0)      0
   3     6  26.7%    18      3 (16.7%)        12 (15|3)     0
   1     1   2.9%   375      8 (2.1%)          0 (0|0)      1
   6     2   6.5%   408     29 (7.1%)          0 (0|0)      1
  13     2   7.0%    26      2 (7.7%)          0 (0|0)      1

Tested 868, errors 47 (5.4%)   <<


        (a)   (b)   <-classified as
       ----  ----
        801     3   (a): class 1
         44    20   (b): class 0
```

The 12th Model: Model12

```
C4.5 [release 8] rule generator     Fri Jul 27 03:22:35 2001
-------------------------------

    Options:
      File stem <jigsaw10_her_comp>

Read 744 cases (12 attributes) from jigsaw10_her_comp

------------------
Processing tree 0

Final rules from tree 0:

Model Model12 :

Rule 3:
      NON > 0
      MDS > 27
      DEPCC <= 9
      ->  class 0  [50.0%]

Rule 5:
      NON > 0
      WMC <= 107
      WMCLOC > 434
      ->  class 0  [50.0%]

Rule 1:
      MDS <= 27
      ->  class 1  [99.3%]

Rule 2:
      NON <= 0
      ->  class 1  [98.2%]

Default class: 1

$

Evaluation on training data (744 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
   3     3  50.0%     2      0 (0.0%)     2 (2|0)   0
   5     3  50.0%     2      0 (0.0%)     2 (2|0)   0
   1     1   0.7%   568      2 (0.4%)     0 (0|0)   1
   2     1   1.8%   159      7 (4.4%)     0 (0|0)   1

Tested 744, errors 9 (1.2%)   <<

         (a)   (b)     <-classified as
        ----  ----
         731           (a): class 1
           9     4     (b): class 0
```

The 13th Model: Mode13

```
C4.5 [release 8] rule generator     Fri Jul 27 03:23:54 2001
-------------------------------

    Options:
      File stem <jigsaw212_her_comp>

Read 371 cases (14 attributes) from jigsaw212_her_comp

------------------
Processing tree 0

Final rules from tree 0:

Model Model13 :

Rule 2:
      NPPM <= 17
      ->  class 1  [99.6%]

Rule 1:
      NPA <= 8
      ->  class 1  [99.6%]

Rule 4:
      DIT > 1
      ->  class 1  [99.3%]

Rule 3:
      NON <= 0
      DIT <= 1
      NPA > 8
      NPPM > 17
      ->  class 0  [50.0%]

Default class: 1

$

Evaluation on training data (371 items):

Rule  Size  Error  Used  Wrong                  Advantage
----  ----  -----  ----  -----                  ---------
   2     1   0.4%   331      0 (0.0%)      0 (0|0)   1
   1     1   0.4%    22      0 (0.0%)      0 (0|0)   1
   4     1   0.7%    14      0 (0.0%)      0 (0|0)   1
   3     4  50.0%     2      0 (0.0%)      2 (2|0)   0

Tested 371, errors 0 (0.0%)   <<

         (a)   (b)     <-classified as
        ----  ----
         369           (a): class 1
                2      (b): class 0
```

The 14th Model: Model14

```
C4.5 [release 8] rule generator     Sat Jul 28 20:47:49 2001
-------------------------------

    Options:
      File stem <jigsaw212_coh_coup>

Read 947 cases (10 attributes) from jigsaw212_coh_coup

------------------
Processing tree 0

Final rules from tree 0:

Model Model14 :

Rule 4:
      coh > 0.13095
      OCMAIC > 6
      NOC > 1
      ->  class 0  [50.0%]

Rule 2:
      OCMAIC <= 6
      ->  class 1  [99.5%]

Rule 1:
      NOC <= 1
      ->  class 1  [99.3%]

Rule 3:
      coh <= 0.13095
      ->  class 1  [99.3%]

Default class: 1

$

Evaluation on training data (947 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   4     3  50.0%     2      0 (0.0%)          2 (2|0)      0
   2     1   0.5%   776      2 (0.3%)          0 (0|0)      1
   1     1   0.7%   157      2 (1.3%)          0 (0|0)      1
   3     1   0.7%    12      0 (0.0%)          0 (0|0)      1

Tested 947, errors 4 (0.4%)   <<

        (a)   (b)  <-classified as
       ----  ----
        941         (a): class 1
          4     2   (b): class 0
```

The 15th Model: Model15

```
C4.5 [release 8] rule generator     Fri Jul 27 03:29:38 2001
-------------------------------

    Options:
      File stem <voji06_coh_coup>

Read 31 cases (10 attributes) from voji06_coh_coup

------------------
Processing tree 0

Final rules from tree 0:

Model Model15 :

Rule 1:
      COM <= 0
      ->  class 1  [94.4%]

Rule 3:
      OCMAIC > 1
      ->  class 1  [92.2%]

Rule 2:
      COM > 0
      OCMAIC <= 1
      ->  class 0  [31.4%]

Default class: 1

$


Evaluation on training data (31 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   1     1   5.6%    24      0 (0.0%)          0 (0|0)      1
   3     1   7.8%     4      0 (0.0%)          0 (0|0)      1
   2     2  68.6%     3      1 (33.3%)         1 (2|1)      0

Tested 31, errors 1 (3.2%)   <<


        (a)  (b)  <-classified as
       ---- ----
         28    1  (a): class 1
               2  (b): class 0
```

The 16th Model: Model16

```
C4.5 [release 8] rule generator     Fri Jul 27 03:30:25 2001
-------------------------------

    Options:
      File stem <voji06_her_comp>

Read 31 cases (14 attributes) from voji06_her_comp

------------------
Processing tree 0

Final rules from tree 0:

Model Model16 :

Rule 1:
      NOM <= 7
      ->  class 1  [94.8%]

Rule 3:
      MDS > 8
      ->  class 1  [91.2%]

Rule 2:
      MDS <= 8
      NOM > 7
      ->  class 0  [50.0%]

Default class: 1

$


Evaluation on training data (31 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   1     1   5.2%    26      0 (0.0%)          0 (0|0)      1
   3     1   8.8%     3      0 (0.0%)          0 (0|0)      1
   2     2  50.0%     2      0 (0.0%)          2 (2|0)      0

Tested 31, errors 0 (0.0%)   <<


        (a)   (b)  <-classified as
       ----  ----
         29        (a): class 1
               2   (b): class 0
```

The 17th Model: Model17

```
C4.5 [release 8] rule generator     Fri Jul 27 03:30:25 2001
-------------------------------

    Options:
      File stem <voji06_her_comp>

Read 31 cases (14 attributes) from voji06_her_comp

------------------
Processing tree 0

Final rules from tree 0:


C4.5 [release 8] rule generator     Fri Jul 27 03:31:05 2001
-------------------------------

    Options:
      File stem <voji06_coup_her>

Read 31 cases (11 attributes) from voji06_coup_her

------------------
Processing tree 0

Final rules from tree 0:

Model Model17 :

Rule 2:
      DIT > 1
      ->  class 1  [92.2%]

Rule 1:
      OCMAIC <= 2
      DIT <= 1
      MDS > 6
      ->  class 0  [50.0%]

Default class: 1

$

Evaluation on training data (31 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
   2     1   7.8%    17      0 (0.0%)      0 (0|0)   1
   1     3  50.0%     2      0 (0.0%)      2 (2|0)   0

Tested 31, errors 0 (0.0%)   <<

         (a)  (b)     <-classified as
        ----  ----
          29          (a): class 1
                2     (b): class 0
```

The 18th Model: Model18

```
C4.5 [release 8] rule generator     Fri Jul 27 03:32:10 2001
-------------------------------

    Options:
      File stem <voji06_coh_comp>

Read 31 cases (11 attributes) from voji06_coh_comp

------------------
Processing tree 0

Final rules from tree 0:

Model Model18 :

Rule 1:
      NOM <= 7
      ->  class 1  [94.8%]

Rule 2:
      NOM > 7
      NOM <= 8
      ->  class 0  [50.0%]

Default class: 1

$

Evaluation on training data (31 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   1     1   5.2%    26      0 (0.0%)          0 (0|0)      1
   2     2  50.0%     2      0 (0.0%)          2 (2|0)      0

Tested 31, errors 0 (0.0%)   <<


        (a)  (b)  <-classified as
       ---- ----
         29       (a): class 1
               2  (b): class 0
```

The 19th Model: Model19

```
C4.5 [release 8] rule generator     Sat Jul 28 20:48:31 2001
-------------------------------

    Options:
      File stem <jigsaw212_coh_coup_her>

Read 947 cases (15 attributes) from jigsaw212_coh_coup_her

------------------
Processing tree 0

Final rules from tree 0:

Model Model19 :

Rule 2:
      LCOMB > 84
      DIT <= 1
      ->  class 0  [31.4%]

Rule 5:
      coh <= 0.20789
      NOC > 1
      DIT > 6
      ->  class 0  [31.4%]

Rule 1:
      LCOMB <= 84
      DIT <= 6
      ->  class 1  [99.7%]

Rule 4:
      NOC <= 1
      ->  class 1  [99.3%]

Default class: 1

$

Evaluation on training data (947 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   2     2  68.6%     3      1 (33.3%)         1 (2|1)      0
   5     3  68.6%     3      1 (33.3%)         1 (2|1)      0
   1     2   0.3%   867      1 (0.1%)          0 (0|0)      1
   4     1   0.7%    65      1 (1.5%)          0 (0|0)      1

Tested 947, errors 4 (0.4%)   <<

        (a)   (b)  <-classified as
       ----  ----
        939     2  (a): class 1
          2     4  (b): class 0
```

The 20th Model: Model20

```
C4.5 [release 8] rule generator     Sat Jul 28 20:49:09 2001
-------------------------------

    Options:
      File stem <jigsaw212_coh_her>

Read 947 cases (11 attributes) from jigsaw212_coh_her
------------------
Processing tree 0

Final rules from tree 0:

Model Model20 :

Rule 2:
      LCOMB > 84
      DIT <= 1
      ->  class 0  [31.4%]

Rule 5:
      coh <= 0.20789
      NOC > 1
      DIT > 6
      ->  class 0  [31.4%]

Rule 1:
      LCOMB <= 84
      DIT <= 6
      ->  class 1  [99.7%]

Rule 4:
      NOC <= 1
      ->  class 1  [99.3%]

Default class: 1

$

Evaluation on training data (947 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   2     2  68.6%     3      1 (33.3%)         1 (2|1)      0
   5     3  68.6%     3      1 (33.3%)         1 (2|1)      0
   1     2   0.3%   867      1 (0.1%)          0 (0|0)      1
   4     1   0.7%    65      1 (1.5%)          0 (0|0)      1

Tested 947, errors 4 (0.4%)   <<

        (a)   (b)  <-classified as
       ----  ----
        939     2  (a): class 1
          2     4  (b): class 0
```

The 21st Model: Model21

```
C4.5 [release 8] rule generator     Sat Jul 28 20:51:01 2001
-------------------------------
    Options:
      File stem <jigsaw212_her_comp>

Read 371 cases (14 attributes) from jigsaw212_her_comp
------------------
Processing tree 0

Final rules from tree 0:

Model Model21 :

Rule 2:
      NPPM <= 17
      ->  class 1  [99.6%]

Rule 1:
      NPA <= 8
      ->  class 1  [99.6%]

Rule 4:
      DIT > 1
      ->  class 1  [99.3%]

Rule 3:
      NON <= 0
      DIT <= 1
      NPA > 8
      NPPM > 17
      ->  class 0  [50.0%]

Default class: 1

$

Evaluation on training data (371 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   2     1   0.4%   331      0 (0.0%)          0 (0|0)      1
   1     1   0.4%    22      0 (0.0%)          0 (0|0)      1
   4     1   0.7%    14      0 (0.0%)          0 (0|0)      1
   3     4  50.0%     2      0 (0.0%)          2 (2|0)      0

Tested 371, errors 0 (0.0%)   <<

        (a)   (b)  <-classified as
       ----  ----
        369         (a): class 1
              2     (b): class 0
```

The 22nd Model: Model22

```
C4.5 [release 8] rule generator     Tue Jul 31 01:29:40 2001
-------------------------------

    Options:
      File stem <jigsaw205_her_comp>

Read 868 cases (14 attributes) from jigsaw205_her_comp

------------------
Processing tree 0

Final rules from tree 0:

Model Model22 :

Rule 3:
      DIT <= 2
      MDS > 14
      MDS <= 17
      NPA <= 0
      DEPCC <= 20
      ->  class 0  [92.6%]

Rule 20:
      NON <= 0
      CHM <= 37
      NPA > 2
      DEPCC > 20
      ->  class 0  [75.8%]

Rule 16:
      NOM > 5
      NPA > 0
      NPPM <= 4
      ->  class 0  [63.0%]

Rule 23:
      NON <= 0
      DIT <= 4
      DEPCC > 36
      DEPCC <= 44
      ->  class 0  [63.0%]

Rule 8:
      NOC > 5
      DIT <= 1
      ->  class 0  [50.0%]

Rule 15:
      DIT > 2
      NOM <= 5
      ->  class 1  [97.5%]
```

Continue in next page

Continued from last page

```
Rule 2:
      MDS <= 14
      NPA <= 0
      ->  class 1  [96.5%]

Rule 24:
      NON <= 0
      DIT > 4
      ->  class 1  [96.0%]

Rule 25:
      NON > 0
      ->  class 1  [95.3%]

Rule 11:
      DIT > 1
      NPA > 0
      DEPCC <= 20
      ->  class 1  [94.9%]

Rule 7:
      NOC <= 5
      DIT <= 1
      DEPCC <= 20
      ->  class 1  [94.6%]

Default class: 1

$

Evaluation on training data (868 items):

Rule  Size  Error  Used  Wrong                Advantage
----  ----  -----  ----  -----                ---------
   3     5   7.4%    18      0 (0.0%)         18 (18|0)      0
  20     4  24.2%     5      0 (0.0%)          5 (5|0)       0
  16     3  37.0%     3      0 (0.0%)          3 (3|0)       0
  23     4  37.0%     3      0 (0.0%)          3 (3|0)       0
   8     2  50.0%     2      0 (0.0%)          2 (2|0)       0
  15     2   2.5%   105      1 (1.0%)          0 (0|0)       1
   2     2   3.5%   200      5 (2.5%)          0 (0|0)       1
  24     2   4.0%    51      0 (0.0%)          0 (0|0)       1
  25     1   4.7%    73      1 (1.4%)          0 (0|0)       1
  11     3   5.1%   158      8 (5.1%)          0 (0|0)       1
   7     3   5.4%   225     14 (6.2%)          0 (0|0)       1

Tested 868, errors 33 (3.8%)   <<


        (a)   (b)   <-classified as
       ----  ----
        804         (a): class 1
         33    31   (b): class 0
```

The 23rd Model: Model23

```
C4.5 [release 8] rule generator     Tue Jul 31 01:28:34 2001
-------------------------------

    Options:
      File stem <jigsaw205_coh_coup>

Read 868 cases (10 attributes) from jigsaw205_coh_coup

------------------
Processing tree 0

Final rules from tree 0:

Model Model23 :

Rule 14:
      coh > 0.033735
      COMI > 0.16667
      COMI <= 0.875
      OCMAIC > 10
      ->  class 0  [75.8%]

Rule 3:
      coh <= 0.033735
      COM <= 0.15789
      OCMAIC > 2
      OCMAIC <= 4
      CUBF > 4
      CUBF <= 7
      ->  class 0  [73.3%]

Rule 1:
      OCMAIC <= 2
      ->  class 1  [97.1%]

Rule 6:
      coh > 0.033735
      OCMAIC <= 10
      ->  class 1  [93.5%]

Rule 13:
      coh > 0.033735
      COMI <= 0.16667
      ->  class 1  [93.0%]

Default class: 1

$
```

Continue in next page

Continued from last page

```
Evaluation on training data (868 items):

Rule  Size  Error  Used  Wrong                  Advantage
----  ----  -----  ----  -----                  ---------
  14     4  24.2%     5      0 (0.0%)           5 (5|0)      0
   3     6  26.7%    18      3 (16.7%)         12 (15|3)     0
   1     1   2.9%   375      8 (2.1%)           0 (0|0)      1
   6     2   6.5%   408     29 (7.1%)           0 (0|0)      1
  13     2   7.0%    26      2 (7.7%)           0 (0|0)      1

Tested 868, errors 47 (5.4%)   <<


        (a)   (b)   <-classified as
       ----  ----
        801     3   (a): class 1
         44    20   (b): class 0
```

Appendix B Experiment Results and Combination Models:

No.1 Experiment

- **Data environment**: *Traing_1 & Testing_1.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.69%*
2nd	*5%*	*1*	*71.17%*
3rd	*5%*	*10%*	*70.37%*
4th	*10%*	*1*	*70.69%*
5th	*10%*	*1*	*73.10%*
6th	*10%*	*10%*	*70.69%*
Fitness value of combination model in training dataset: *73.10%*			
Fitness value of combination model in testing dataset: *72.10%*			
Best fitness value of original models in training: *69.08%*			
Best fitness value of original models in testing: *68.23%*			

The best combination model with the fitness value: *73.10%*

```
Rule 011114:
     coh > 0.083735
     COMI > 0.16667
     COMI <= 2.875
     OCMAIC > 12.0
          -> class 1  [75.8%]

Rule 010422:
     DEPCC <= 18.0
          -> class 1  [86.9%]

Rule 010308
     OMAEC > 6.0
     DIT > 1.0
          -> class 1  [71.8%]

Rule 011106
     coh > 0.083735
     OCMAIC <= 13.0
          -> class 0  [93.5%]

Rule 011113
     coh > 0.133735
     COMI <= 0.16667
          -> class 1  [93.0%]

Default class:0
Fitness Value:0.7310789049919485
```

No.2 Experiment :

- **Data environment**: *Traing_2 & Testing_2.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.37%*
2nd	*5%*	*1*	*68.92%*
3rd	*5%*	*10%*	*70.37%*
4th	*10%*	*1*	*72.30%*
5th	*10%*	*1*	*70.69%*
6th	*10%*	*10%*	*70.37%*
Fitness value of combination model in training dataset: *72.30%*			
Fitness value of combination model in testing dataset: *70.30%*			
Best fitness value of original models in training: *69.08%*			
Best fitness value of original models in testing: *66.23%*			

```
The best combination model with the fitness value:
72.30%
          Rule Name: Rule011114
               coh > 0.033735
               COMI > 0.16667
               COMI <= 0.875
               OCMAIC > 10.0
                   -> class 0   [75.8%]

          Rule Name: Rule010325
               CUBF > 21.0
               CHM > 37.0
                   -> class 0   [91.7%]

          Rule Name: Rule010317
               NOP > 11.0
                   -> class 1   [83.3%]

          Rule Name: Rule011106
               coh > 0.083735
               OCMAIC <= 13.0
                   -> class 1   [93.5%]

          Rule Name: Rule011113
               coh > 0.033735
               COMI <= 0.16667
                   -> class 1   [93.0%]

          Default class:1
          Fitness Value:0.7230273752012882
```

No.3 Experiment:

- **Data environment**: *Traing_3 & Testing_3.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.04%*
2nd	*5%*	*1*	*71.49%*
3rd	*5%*	*10%*	*70.37%*
4th	*10%*	*1*	*71.65%*
5th	*10%*	*1*	*70.69%*
6th	*10%*	*10%*	*70.69%*
Fitness value of combination model in training dataset: *71.65%*			
Fitness value of combination model in testing dataset: *71.65%*			
Best fitness value of original models in training: *68.11%*			
Best fitness value of original models in testing: *69.10%*			

The best combination model with the fitness value: *70.65%*

```
Rule 012314:
     coh > 0.033735
     COMI > 0.16667
     COMI <= 0.875
     OCMAIC > 10.0
         -> class 0  [75.8%]

Rule 010325:
     CUBF > 21.0
     CHM > 24.0
         -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
         -> class 1  [83.3%]

Rule 012306:
     coh > 0.083735
     OCMAIC <= 10.0
         -> class 0  [93.5%]

Rule 012313:
     coh > 0.033735
     COMI <= 0.16667
         -> class 1  [93.0%]

Default class: 1
Fitness Value:0.71658615136876
```

No.4 Experiment:

- **Data environment**: *Traing_4 & Testing_4.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*71.49%*
2nd	*5%*	*1*	*71.33%*
3rd	*5%*	*10%*	*71.65%*
4th	*10%*	*1*	*71.81%*
5th	*10%*	*1*	*71.33%*
6th	*10%*	*10%*	*71.65%*
Fitness value of combination model in training dataset: *71.81%*			
Fitness value of combination model in testing dataset: *73.25%*			
Best fitness value of original models in training: *70.37%*			
Best fitness value of original models in testing: *70.22%*			

No.4 Experiment: (*Continued from last page*)

The best combination model with the fitness value: *71.81%*

```
Rule Name: Rule010321
     OCMAIC <= 10.0
     CUBF > 6.0
     CHM > 24.0
         -> class 0  [94.4%]

Rule Name: Rule010423
     NPPM > 16.0
     DEPCC > 2.0
         -> class 0  [79.6%]

Rule Name: Rule010419
     WMC <= 22.0
     MCC > 17.0
     DEPCC > 2.0
         -> class 1  [63.3%]

Rule Name: Rule010313
     CUBF > 9.0
     NOC > 2.0
         -> class 1  [79.4%]

Rule Name: Rule010316
     OMAEC > 1.0
     NOP > 2.0
         -> class 1  [76.1%]

Rule Name: Rule010308
     OMAEC > 6.0
     DIT > 3.0
         -> class 1  [71.8%]

Rule Name: Rule010318
     CUBF <= 6.0
     OMAEC <= 7.0
     NOP <= 1.0
         -> class 0  [91.8%]

Rule Name: Rule010315
     OMAEC <= 1.0
     NOP <= 4.0
     CHM <= 26.0
         -> class 1  [88.9%]

Rule Name: Rule010314
     NOP <= 2.0
     CHM <= 24.0
         -> class 1  [88.4%]

Default class:1
Fitness Value:0.7181964573268921
```

No.5 Experiment:

- **Data environment**: *Traing_5 & Testing_5.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*69.88%*
2nd	*5%*	*1*	*70.04%*
3rd	*5%*	*10%*	*70.20%*
4th	*10%*	*1*	*69.88%*
5th	*10%*	*1*	*70.85%*
6th	*10%*	*10%*	*70.20%*
Fitness value of combination model in training dataset: *70.85%*			
Fitness value of combination model in testing dataset: *70.20%*			
Best fitness value of original models in training: *68.59%*			
Best fitness value of original models in testing: *68.43%*			

No.5 Experiment: (*Continued from last page)*

The best combination model with the fitness value: *70.85%*

```
Rule 010321:
     OCMAIC <= 7.0
     CUBF > 6.0
     CHM > 24.0
          -> class 0  [94.4%]

Rule 010325:
     CUBF > 21.0
     CHM > 24.0
          -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
          -> class 1  [83.3%]

Rule 010313:
     CUBF > 8.0
     NOC > 2.0
          -> class 0  [79.4%]

Rule 010316:
     OMAEC > 1.0
     NOP > 2.0
          -> class 0  [76.1%]

Rule 010308:
     OMAEC > 6.0
     DIT > 1.0
          -> class 0  [71.8%]

Rule 010318:
     CUBF <= 6.0
     OMAEC <= 7.0
     NOP <= 1.0
          -> class 1  [91.8%]

Rule 010315:
     OMAEC <= 1.0
     NOP <= 4.0
     CHM <= 24.0
          -> class 1  [88.9%]

Rule 010314:
     NOP <= 3.0
     CHM <= 24.0
          -> class 1  [88.4%]

Default class: 1
Fitness Value:0.7085346215780999
```

No.6 Experiment:

- **Data environment**: *Traing_6 & Testing_6.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.37%*
2nd	*5%*	*1*	*68.92%*
3rd	*5%*	*10%*	*70.37%*
4th	*10%*	*1*	*72.62%*
5th	*10%*	*1*	*70.69%*
6th	*10%*	*10%*	*70.37%*
Fitness value of combination model in training dataset: 72.62%			
Fitness value of combination model in testing dataset: 70.11%			
Best fitness value of original models in training: 68.76%			
Best fitness value of original models in testing: 64.51%			

The best combination model with the fitness value: *72.65%*

```
Rule 012314:
     coh > 0.033735
     COMI > 0.16667
     COMI <= 0.875
     OCMAIC > 10.0
          -> class 0  [75.8%]

Rule 010325:
     CUBF > 21.0
     CHM > 37.0
          -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
          -> class 1  [83.3%]

Rule 012306:
     coh > 0.083735
     OCMAIC <= 13.0
          -> class 0  [93.5%]

Rule 012313:
     coh > 0.033735
     COMI <= 0.16667
          -> class 1  [93.0%]

Default class: 1
Fitness Value:0.7262479871175523
```

No.7 Experiment:

- **Data environment**: *Traing_7 & Testing_7.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*70.85%*
2nd	*5%*	*1*	*69.883%*
3rd	*5%*	*10%*	*69.88%*
4th	*10%*	*1*	*69.88%*
5th	*10%*	*1*	*69.88%*
6th	*10%*	*10%*	*69.88%*
Fitness value of combination model in training dataset: 70.85%			
Fitness value of combination model in testing dataset: 69.31%			
Best fitness value of original models in training: 68.27%			
Best fitness value of original models in testing: 66.54%			

No.7 Experiment: *(Continued from last page)*

The best combination model with the fitness value: *70.85%*

```
Rule 010321:
     OCMAIC <= 7.0
     CUBF > 6.0
     CHM > 24.0
          -> class 0  [94.4%]

Rule 010325:
     CUBF > 21.0
     CHM > 24.0
          -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
          -> class 1  [83.3%]

Rule 010313:
     CUBF > 8.0
     NOC > 3.0
          -> class 1  [79.4%]

Rule 010316:
     OMAEC > 1.0
     NOP > 2.0
          -> class 1  [76.1%]

Rule 010308:
     OMAEC > 6.0
     DIT > 3.0
          -> class 1  [71.8%]

Rule 010318:
     CUBF <= 6.0
     OMAEC <= 7.0
     NOP <= 2.0
          -> class 1  [91.8%]

Rule 010315:
     OMAEC <= 1.0
     NOP <= 4.0
     CHM <= 24.0
          -> class 1  [88.9%]

Rule 010314:
     NOP <= 2.0
     CHM <= 24.0
          -> class 1  [88.4%]

Default class: 0
Fitness Value:0.7085346215780999
```

No.8 Experiment:

- **Data environment**: *Traing_8 & Testing_8.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*69.88%*
2nd	*5%*	*1*	*69.88%*
3rd	*5%*	*10%*	*70.04%*
4th	*10%*	*1*	*69.88%*
5th	*10%*	*1*	*70.04%*
6th	*10%*	*10%*	*69.88%*
Fitness value of combination model in training dataset: 70.04%			
Fitness value of combination model in testing dataset: 68.12%			
Best fitness value of original models in training: 68.72%			
Best fitness value of original models in testing: 67.12%			

No.8 Experiment:*(Continued from last page)*

The combination model with the best fitness value: *70.04%*

```
Rule 010321:
     OCMAIC <= 10.0
     CUBF > 6.0
     CHM > 24.0
          -> class 0  [94.4%]

Rule 010325:
     CUBF > 21.0
     CHM > 24.0
          -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
          -> class 1  [83.3%]

Rule 010308:
     OMAEC > 6.0
     DIT > 3.0
          -> class 1  [71.8%]

Rule 010318:
     CUBF <= 6.0
     OMAEC <= 7.0
     NOP <= 1.0
          -> class 1  [91.8%]

Rule 010315:
     OMAEC <= 1.0
     NOP <= 4.0
     CHM <= 24.0
          -> class 1  [88.9%]

Rule 010314:
     NOP <= 3.0
     CHM <= 24.0
          -> class 1  [88.4%]

Default class: 1
Fitness Value:0.7004830917874396
```

No.9 Experiment:

- **Data environment**: *Traing_9 & Testing_9.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*71.65%*
2nd	*5%*	*1*	*72.14%*
3rd	*5%*	*10%*	*72.30%*
4th	*10%*	*1*	*71.65%*
5th	*10%*	*1*	*71.65%*
6th	*10%*	*10%*	*72.14%*
Fitness value of combination model in training dataset: 72.30%			
Fitness value of combination model in testing dataset: 72.30%			
Best fitness value of original models in training: 70.04%			
Best fitness value of original models in testing: 69.12%			

No.9 Experiment:*(Continued from last page)*

The best combination model with the fitness value:*72.30%*

```
Rule 010321:
     OCMAIC <= 7.0
     CUBF > 6.0
     CHM > 24.0
          -> class 0  [94.4%]

Rule 010325:
     CUBF > 21.0
     CHM > 24.0
          -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
          -> class 1  [83.3%]

Rule 010313:
     CUBF > 8.0
     NOC > 2.0
          -> class 1  [79.4%]

Rule 010316:
     OMAEC > 1.0
     NOP > 2.0
          -> class 1  [76.1%]

Rule 010308:
     OMAEC > 6.0
     DIT > 2.0
          -> class 0  [71.8%]

Rule 010318:
     CUBF <= 6.0
     OMAEC <= 7.0
     NOP <= 3.0
          -> class 1  [91.8%]

Rule 010315:
     OMAEC <= 1.0
     NOP <= 4.0
     CHM <= 24.0
          -> class 1  [88.9%]

Rule 010314:
     NOP <= 4.0
     CHM <= 24.0
          -> class 1  [88.4%]

Default class: 0
Fitness Value:0.7230273752012882
```

No.10 Experiment:

- **Data environment**: *Traing_10 & Testing_10.*
- **Number of generation**: *100*
- **Maximum population size in a generation**: *160.*
- **Crossover probability**: *0.8*

Iteration	Mutation Probability	Elitist	Best Fitness Value
1st	*5%*	*1*	*71.33%*
2nd	*5%*	*1*	*70.04%*
3rd	*5%*	*10%*	*70.04%*
4th	*10%*	*1*	*70.37%*
5th	*10%*	*1*	*70.04%*
6th	*10%*	*10%*	*70.04%*
Fitness value of combination model in training dataset: 71.33%			
Fitness value of combination model in testing dataset: *70.37%*			
Best fitness value of original models in training: 68.43%			
Best fitness value of original models in testing: *70.37%*			

No.10 Experiment: *(continued from last page)*

The best combination model with the fitness value: *71.33%*

```
Rule 010321:
     OCMAIC <= 7.0
     CUBF > 6.0
     CHM > 24.0
         -> class 0  [94.4%]

Rule 010325:
     CUBF > 21.0
     CHM > 24.0
         -> class 0  [91.7%]

Rule 010317:
     NOP > 4.0
         -> class 1  [83.3%]

Rule 010318:
     CUBF <= 6.0
     OMAEC <= 7.0
     NOP <= 1.0
         -> class 1  [91.8%]

Rule 010315:
     OMAEC <= 1.0
     NOP <= 4.0
     CHM <= 24.0
         -> class 1  [88.9%]

Rule 010314:
     NOP <= 2.0
     CHM <= 24.0
         -> class 1  [88.4%]

Default class: 1
Fitness Value:0.7133655394524959
```